The Finger Walker: A Method to Navigate Virtual Environments

By

Sanford Brian Fitch

Submitted to the Department of Electrical Engineering and Computer Science
In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and Master of Engineering in Electrical Engineering and Computer Science
at the
Massachusetts Institute of Technology
May 26, 1998

Author_____
Department of Electrical Engineering and Computer Science
May 26, 1998

Certified by_____
Nathaniel Durlach
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

The Finger Walker: A Method to Navigate Virtual Environments

By

Sanford Brian Fitch

Submitted to the
Department of Electrical Engineering and Computer Science

May 26, 1998

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

There are many factors which make virtual environment (VE) systems particularly useful for training applications. Not only can VE systems be easily reconfigured to simulate different real situations, but they can be used to create situations that could not exist in the real world but nonetheless are exceptionally effective in training. Within the general training area, work in this thesis focuses on training directed towards the acquisition of spatial knowledge. There are many cases in which spatial knowledge cannot be acquired in the actual environment, and the training must be accomplished by other means using a VE. A critical factor contributing to the acquisition of spatial knowledge is the method employed for moving around within the VE. Some methods of movement do not provide the user with any easily sensed measure of the amount of effort or work that would be associated with the movement in the real world.

This thesis concentrates on the development of an interface that enables the user to "finger walk" through a VE. This interface makes use of a low friction pad that allows the user to finger walk "in place" and an electric field sensing system that monitors the position of the fingers on the pad. The user interface designed effectively tracks the user's movement along the surface of the pad for input into a VE.

Thesis Supervisor: Nathaniel I. Durlach
Title: Senior Research Scientist
M.I.T. Research Laboratory of Electronics

Contents

Figures

Tables

1 Introduction

Virtual environments are receiving attention by experts in many fields, including, Computer Science, Electrical Engineering, Mechanical Engineering, Psychology, Architecture, and Medicine. In a virtual environment system, a human interacts with a computer-generated environment via a human-machine interface. The interface contains devices that display computer generated information to the human user (displays) and devices that sense the state of the user and are used to control the actions of the computer (controls). Display devices may address any sensory modality. For example, there are visual displays (e.g., monitors), auditory displays (e.g., earphones), and tactile displays (e.g., vibratory arrays). Similarly, control devices may sense various types of motor activities (e.g., joysticks or data gloves to sense manual actions or speech recognition systems to sense articulatory output) or neural activities (e.g., electrodes to sense brain-wave activity).

In contrast to teleoperator systems, the purpose of which is to enhance the human operator's ability to sense, travel through, or modify the real world, "the purpose of virtual-environment systems is to sense, manipulate, and transform the state of the human operator (as in training or scientific visualization) or to modify the state of the information stored in the computer (e.g., the virtual environment or some theoretical model represented in the computer software)" (Durlach and Mavor, 1995, p.19). Depending on the desired application, a virtual environment might closely approximate reality or create an imaginary world that differs radically from the real world.

The definition of a virtual environment system and the relation of virtual environment systems to classical simulators and teleoperator systems are discussed in detail

in Durlach and Mavor (1995). Potential application areas include manufacturing, information visualization, health care, teletravel, marketing, entertainment, and training. This thesis is focused on training applications.

There are many factors which make virtual environment systems particularly useful for training applications. Advantages in the areas of cost and safety are well illustrated by the past and present use of simulation in the training of individuals to pilot airplanes, drive land vehicles, or handle ships and submarines. Such advantages are also becoming increasingly evident in the training of maintenance personnel and medical staff (e.g., surgeons). In addition to the advantages of decreased cost and increased safety, virtual-environment systems provide great flexibility in changing the training paradigm. Not only can these systems be easily reconfigured to simulate different real situations, but they can be used to create situations that could not exist in the real world but nevertheless are exceptionally effective in training (e.g., by unrealistically emphasizing certain training goals or by continuously adapting the training environment to the state of the individual user).

Within the general training area, work in this thesis focuses on training directed towards the acquisition of spatial knowledge. There are many cases in which spatial knowledge cannot be acquired in the actual environment (i.e. military special forces often do not have access to the space in which they must carry out their mission), and the training must be accomplished by other means using a virtual environment.

This thesis concentrates on the development of an interface that enables the user to "finger walk" through a virtual environment. This interface will make use of a slippery pad that allows the user to finger walk "in place" and a sensing system that monitors the position of the fingers on the pad.

The potential benefits of this work are twofold. First, it is possible that many of the expected advantages of a walking interface can be realized in a more cost-effective manner by means of a scaled-down, finger-walking interface. Secondly, even if this is not the case, experience gained in developing the slippery-pad, finger-walking interface may be useful for later work on a slippery-floor walking interface.

The goal of this thesis is to describe the design of an inexpensive method of navigation which simulates walking within a virtual environment. Chapter 2 further documents some of the goals of this project, such as the use of virtual environments for training spatial knowledge acquisition. In addition, this chapter also discusses several previous input devices and their evolution towards methods which use the effort expended by users to increase their presence in the virtual environment. Chapter 3 describes many of the possible methods commonly used to track the motion for input devices. Chapter 4 details the concept of Electric Field Sensing as a method to quickly and accurately track the motion of a user. Chapter 5 details a performance study involving receiver and transmitter design and implementation. Chapter 6 discusses different methods of determining the position of the user's fingers on the surface of the pad. Chapter 7 details the methods used to determine the magnitude and direction of the user's movement through the virtual environment. Chapter 8 describes the actual design of a finger walker, including both the hardware and software required. Chapter 9 details the development of a graphical user interface (GUI) to test the benefits of the finger walking user interface developed in chapter 8. Finally, chapter 10 concludes with a discussion of the results of tests on the finger walker interface, including a discussion of its possible expansion to a full-scale walking input device.

2 Background/Concepts

The following subsections briefly explain some of the concepts and past experiments which have lead to the concept of a finger walking interface. The concepts of spatial knowledge acquisition, presence and distance estimation, and work in virtual environments are discussed in subsections 2.1, 2.3, and 2.4, while subsection 2.2 gives an overview of interfaces in general, including several devices directly related to a finger-walking interface.

2.1 Spatial Knowledge Acquisition

Within the general training area, work in this thesis focuses on training directed towards the acquisition of spatial knowledge. There are two types of spatial knowledge: route following and configurational knowledge. Typically, route following refers to a person's ability to move from one location to another by following a specific path. Often the person navigating along the path has no knowledge of any alternate path or even perhaps of how to traverse the path in reverse. On the other hand, configurational knowledge refers to a person's knowledge concerning the structural layout of the whole environment. In general, an individual with good configurational knowledge is capable of route following and of determining alternate routes between any two points.

One critical component of training spatial knowledge using a virtual environment system concerns the method employed for moving around within the VE. The easiest and most common methods of movement within VEs is accomplished through the use of hand movements. By simply moving a joystick in a certain direction or a data glove into a

certain posture, the user instructs the computer program that he wishes to move in a particular direction within the virtual environment. These methods are inexpensive and easy to incorporate into a virtual environment. However, they differ from normal locomotion (i.e., walking) in a number of ways. In particular, they do not provide the user with any easily sensed measure of the amount of effort or work that would be associated with the movement in the real world. For example, in the real world, a person walking through a room acquires certain spatial knowledge (e.g., related to the room's length) from the effort exerted in walking through the room. However, this spatial knowledge is lost when using a joystick or data glove to move through the virtual room because the mechanical work associated with the manual action is not related to the work involved in actually walking through the room. By creating a method for movement through a virtual environment which simulates the work involved in walking in the real world, users may gain a better feel for the distance they have traveled in the virtual environment.

2.2 Locomotion Interfaces

In general, mobility, locomotion, or walking interfaces are needed for many tasks other than training spatial knowledge. A recent review of some of the interfaces of this type that have been developed is available in Jacobsen et al (1998). Most of these interfaces, such as the omnidirectional treadmill, not only suffer from a variety of performance deficiencies, but are very complex, bulky, and expensive. The following paragraphs comment briefly on some previously documented results on locomotion interfaces that are particularly relevant to the proposed work.

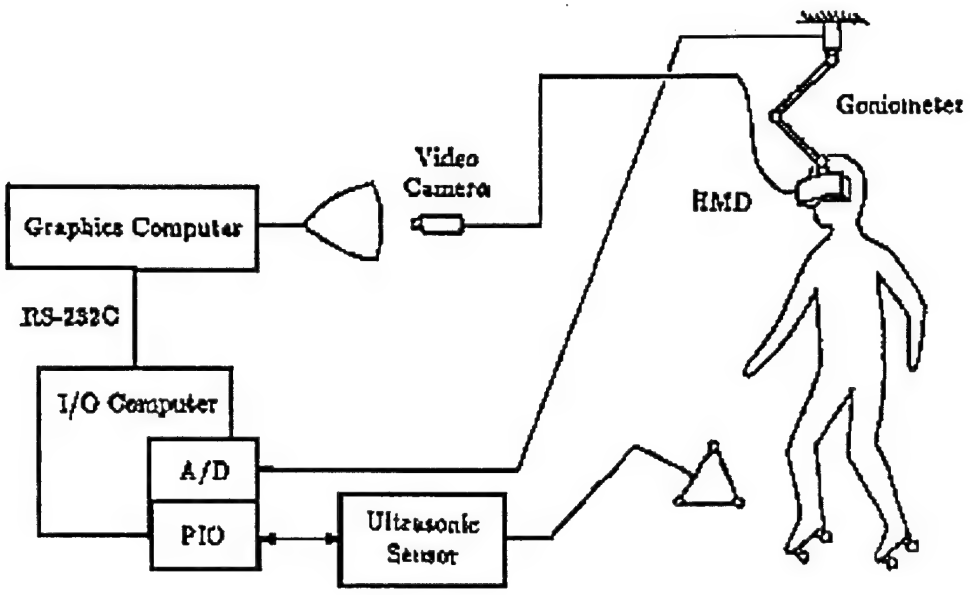


Figure 1: Hardware configuration of Iwata's Haptic Walkthrough Simulator.

Iwata and his associates at the University of Tsukuba in Japan (Iwata, 1992; Iwata and Matsuda, 1992; and Iwata and Fujii, 1996) have performed research relating to interfaces which require the user to walk in place in the real world in order to move through the VE. In one very complex system (Iwata, 1992, Iwata and Matsuda, 1992; see figure 1), the walker was outfitted with specially designed roller skates that enable motion in two dimensions, a harness that fixed the trunk of the walker relative to the overall framework of the system, ultrasonic sensors that measure the positions of the feet and head, a cable system attached to the skates to provide force feedback to the feet, a head mounted display (HMD) to provide visual images, and a 6 degree of freedom haptic force feedback device for the hand. In the experiments performed, subjects were required to estimate distances traversed within the VE in two cases: (1) moving through the VE by means of the walking interface and (2) moving through the VE by means of hand-gesture-controlled flying.

Although these experiments appear to have been performed very crudely, and the results of these experiments are described very briefly, they do suggest that the walking interface led to less response bias than the hand-controlled-flying interface. In other words, test subjects who walked through the test space had a better knowledge of the distance traveled in the VE than subjects flying through the VE. More specifically, the underestimation of distance that appeared for the larger distances was less pronounced when the walking interface was used. No results were presented on resolution in the distance estimation task, only on bias.

In a later prototype developed by Iwata and Fujii (Iwata and Fujii, 1996; see figure 2), the roller skates were replaced by sandals with a low friction film on the sole (together with a break pad), the support harness was replaced by a hoop around the waist, and the

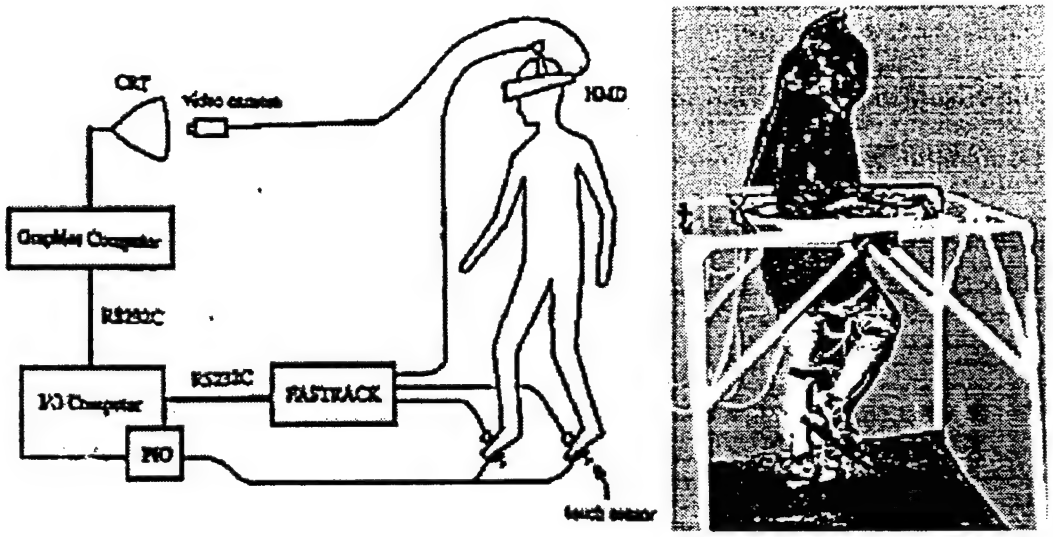


Figure 2: Hardware configuration of Iwata's Virtual Perambulator.

ultrasonic sensors were replaced by Polhemus FASTRACK magnetic sensors. Also, touch sensors were incorporated in the soles of the sandals in order to signal the foot steps of the user to the computer.

Reported tests of this system consisted of observing the extent to which novice users (at SIGGRAPH) were able to walk and turn using this system. According to relatively crude criteria, the results were successful. Test subjects were able to adapt quickly to the new device (94% of the subjects tested were able to adapt to the walking device) walking through the VE, turning, and even running (8% of the subjects tested were able to run in the device).

James Templeman and his colleagues at the Naval Research Lab developed a motion platform, the Gaiter, which allows users to travel through virtual environments by taking virtual steps (Templeman, 1998). This locomotion interface, consisting of position trackers (such as a Polhemus system) attached to the knees and pressure sensors attached under the ball and heel of each foot, monitor the human gait. By rocking his knees in different patters, the user can control his virtual motion. For example, by rocking his knees forward then back, the user moves forward through the virtual environment. Thus, the users hands are completely free from the walking interface, allowing them to be used for other tasks.

Though the Gaiter system has yet to be carefully tested, it is obvious that it suffers to some extent from its lack of naturalness. The knee rocking is not very similar to the natural walking motion of a human. The user must learn the motion patterns recognized by the interface in order to user the Gaiter effectively.

2.3 Presence and Distance Estimation

Studies related to the effect of walking in place on the sense of presence and on distance estimation have been conducted by Slater, Usoh, and Steed (1995), and Witmer and Kline (1998). The "sense of presence" refers to the sense of actually being present in the virtual world. A detailed discussion of the sense of presence can be found in Slater, 1997. Distance estimation relates to a users ability to judge both the distance traveled within the VE and, also, the distance to objects in the VE. One of the key factors effecting a users sense of presence and ability to accurately estimate distances is the user interface employed.

In the study by Slater, Usoh, and Steed, it was determined that presence is increased when using a walking interface in place of a pointing device. A neural net acting on head movement information (obtained from a Polhemus ISOTRAK) was used to determine whether or not the subject was walking in place (no special floor was used). When walking was judged to occur, the direction of movement through the virtual space was determined by gaze direction. In comparison to movement achieved by pointing with a mouse, this walking-in-place method provided a greater sense of presence in the VE, provided that the user identified sufficiently with the virtual body (VB) included in the VE. Whereas for the walking group the degree of presence increased with the degree of identification with the VB, for the pointing group there was no correlation between the two variables.

In the study by Witmer and Kline, distance estimation was examined in the real world and in VEs for both static (subject fixed) and dynamic (subjects moved through the environment) situations using the psychophysical experimentation technique of magnitude estimation. Furthermore, in the dynamic case, three methods of movement were employed

in the VE: joystick, treadmill, and passive teleportation. The results show that subjects underestimate distance and that the effect is more pronounced in VEs than in the real world (at least for the VEs considered). Also, the sense of presence was found to be greater with the treadmill than with the other movement methods. However, although traversing a distance was found to improve distance estimation, the treadmill was not found to be superior to the other movement methods.

Clearly, much more research is needed to determine the manner in which resolution and bias in distance estimation is influenced by the characteristics of the VE and by the characteristics of the movement method selected for exploring the VE. Also, of course, information must be obtained on how such characteristics influence aspects of spatial perception other than distance estimation (for example, angular estimation, or, at a more general level, estimation of topological properties of the space).

2.4 Expending Effort

Finally, it is worth commenting briefly on the importance of perceived effort or work cues related to actions performed with the hand and on possible haptic interfaces for moving through VEs, including the envisioned finger-walking interface. The fact that work cues are important in haptic perception has been clearly demonstrated by Tan and her associates at MIT in their studies of how humans discriminate material properties such as compliance when grasping an object (e.g., Tan et al., 1995). Even when subjects are clearly instructed (as well as provided with correct-response-trial-by-trial-feedback) to judge compliance when squeezing the object, they tend to judge the work done in squeezing the object. It is conceivable that any of a wide range of haptic interfaces in which movement through the VE is controlled by incremental steps involving effort or

work would be useful. Thus, for example, a joystick interface that was programmed so that movement is achieved by repeated throws of the joystick (each throw corresponding to a step) would be just as effective as a finger-walking interface. Whether or not the effort required results from forces associated solely with free-space movement of the hand or from force-feedback devices (e.g., perhaps the joystick stepper should have a spring-return-to-center) may not matter. Of all the possible haptic-step interfaces that one might consider, the finger walker was chosen as the focus of this thesis because of its relative naturalness and because of its natural extendibility to walking interfaces.

3 Tracking Systems

A major component of the proposed finger-walking interface is the tracking system used to sense the position of the fingers. Also, because long term goals include the extension of this interface to a real (leg) walking interface, and because there are many applications in which leg-walking interfaces would have increased utility if they could accommodate such actions as crawling, a key consideration in the selection of a tracking technology for this thesis is its extendibility in terms of both size and method of movement.

Another key consideration for the proposed tracking interface is cost. There are many sensing techniques which seem well suited to position tracking, but when expanded from the small-scale finger walker to the large-scale leg walker become prohibitively expensive. For example, the dental industry uses a device to determine the pressure of a patient's teeth on a pad which provides extremely high resolution and speed at the cost of a few thousand dollars. This type of pressure pad might be suitable for the finger walker; however, when expanded to the leg walker the cost would increase to a few hundred thousand dollars.

A further key consideration in the design of the tracking system is simplicity. Factors contributing to the simplicity of a design include not only the construction and operation of the tracking system, but also the computational power required to translate the output of the tracking system (typically an analog voltage) into a movement vector. For example, machine vision techniques might be useful for tracking multiple-user postures with a video camera. However, the algorithms required to process the images captured by the video camera are too complex to be accomplished in real time without an extremely powerful computer.

Finally, the tracking system needs to coexist with other devices. Mechanical interference can occur when the user is encumbered by large amounts of equipment and gets tangled in the devices. Electrical interference can occur between the tracker and any of the multitude of devices used in VE applications, such as a head mounted display, a tracker, and a haptic interface. The tracking system must be designed without detailed knowledge of the components making up the rest of the human-machine interface.

With these design goals in mind (adaptability, cost, simplicity, and coexistence), plausible methods of motion tracking for a finger walker include: mechanical trackers, pressure-based trackers, photo-cell and fiber-optic trackers, direct-vision trackers, inertial-sensing trackers, ultrasonic trackers, electromagnetic-field trackers, and electric-field trackers. Each of these methods is considered briefly in the following subsections.

3.1 Mechanical Trackers

In mechanical trackers, a system of levers and pulleys is attached to rotary or optical shaft encoders (such as potentiometers) that create an analog or digital signal which can be used to encode the user's position, velocity, and acceleration. This method of tracking is exemplified by the Shooting Star head tracker. A finger-walker tracking system based on this concept would have several advantages; however, it would restrict the user's movements and might lead to mechanical interference.

3.2 Pressure-Based Trackers

A force-sensitive pad is another method for tracking a user's finger movement across an area. Force detection methods involving transducers such as piezoelectric load cells or a network of resistors located in a pad (the resistors are located in such a way that when force is applied to the pad an electrical connection is made; the resistance of the network varies depending on the distance between the contact point and the origin) have a good frequency response and large dynamic range, but suffer from other limitations. For example, piezoelectric load cells lose sensitivity after extended use. Similarly, when resistor networks are used, only two x and two y coordinates can be determined. Another problem is that there is no way to precisely match the correct x and y position coordinates corresponding to the same force. Furthermore, preliminary tests of this type of tracking device showed that the material used in the pad degrades over time, causing problems with repeatability (Bechwith, Marangoni, and Lienhard, 1993).

3.3 Photocell and Fiber-Optic Trackers

A third plausible method for tracking involves the use of photocells or fiber optics. A finger-walking tracker could be set up like a computer touch screen which uses an array of photocells laid out in a grid to measure the x and y position of screen contact. Alternatively, fiber optic clothing (i.e., pants or a glove) could be created with sensors piped along the surface of the clothing to monitor the changes in optical intensity due to the bending of the fibers. Each of these fibers must then be connected to a hardware system which both generates the laser beam and processes the data. Both these devices would be

simple, but the touch screen approach is too costly and the fiber-optic approach is too cumbersome.

3.4 Direct-Vision Trackers

A direct-vision system would make use of a video camera and a machine-vision algorithm. Such a system could track the user's movements without encumbering the user. However, as mentioned previously, machine-vision algorithms require large processing power and are difficult to perform in real time. Even when reflectors are attached to the body to make the scene easier to parse, the system requires intensive computation.

3.5 Inertial-Sensing Trackers

Inertial-sensing trackers use accelerometers attached to the relevant moving parts of the body. By integrating the acceleration twice, both a velocity vector and a position vector can be determined for each sensor. Such a tracking system suffers almost no latency. However, like many of the previous methods, the user is encumbered by extra equipment. In addition, the filtering and software required to determine a stable velocity is not computationally simple. Although as technology advances accelerometers are becoming both smaller and cheaper, they currently remain expensive and encumbering when multiple body parts must be tracked.

3.6 Ultrasonic Trackers

An ultrasonic finger-walking tracker could be set up in either of two configurations: a "transmitter-receiver" configuration or a "ping-and-listen" configuration. In the former, configuration, a receiver attached to each object being tracked returns a signal each time it receives a ping from the transmitter. The tracking of a single point in three dimensions requires the use of a single transmitter and three receivers. Thus, the time of flight for an array of transmitters and receivers results in a position mapping of the relevant body parts which can be compared to the previous map to determine movement. As with the past methods, this type of ultrasonic system is too encumbering for the user who must wear a suit of receivers. In the ping-and-listen configuration, a transmitter located on the pad transmits a ping which partially reflects off the user. This reflection is received by a sensor mounted with the transmitter in the pad. As before, the time of flight for an array of transmitters and receivers is used to calculate the movement vector of the user. This configuration eliminates the need for any equipment on the user; however, the system is complex computationally and would require an enormous research effort into the orientation of the array of transmitters/receivers, the interference due to multiple reflections, and errors due to shapes of the reflectors (feet, legs), in order to determine how well such a system could work.

3.7 Electromagnetic-Field Trackers

Many trackers (e.g., the Polhemus FASTRACK) used in VE applications employ electromagnetic sensors. This method of tracking is accomplished through the use of a 3-axis magnetic dipole source and 3-axis magnetic dipole detectors. The source varies three perpendicular electromagnetic fields in a known pattern, and the resultant nutating magnetic

vector field is measured by the detector dipoles. The nutating magnetic signal received at the detector can then be processed to yield its position and orientation with respect to the source. Multiple body parts can be tracked simultaneously by using multiple receivers (one on each body part to be tracked). Because the system must cycle through each source/detector set, the rate at which the body position is updated decreases as more transmitters are added. Also, as more sensors are used the likelihood that the frequencies used in the tracking system will interfere with other devices employing electromagnetic fields increases (i.e., HMD typically use a Polhemus trackers). In addition, electromagnetic trackers encounter interference problems from surrounding fields. Although such trackers are used extensively in VE applications, they suffer from interference, they are costly, and they encumber the user.

3.8 Electric-Field Trackers

An electric-field tracker is a simple device which can update a VE with the user's velocity without encumbering the user. The user need only attach a single transmitter to his body (i.e., an electrode connected to the user through a wire attached to the HMD) and the position of his body, specifically the fingers, can be detected by receivers located in the area surrounding the user. Thus, the user is encumbered with only a small electrode, not a full set of transmitters as with some of the previously discussed tracker designs. The receivers detect the strength of the field at specific points. If enough receivers are used, multiple body parts can be tracked in multiple dimensions (Zimmerman, Smith, Paradiso, Allport, and Gershenfeld, 1994; Smith, 1995 and 1996; and Bik, 1997).

4 Detailed Consideration of Electric Field Trackers

"Electric Field sensing refers to a family of noncontact measurements of the human body made with slowly varying electric fields"[Zimmerman, Smith, Paradiso, Allport, and Gershenfeld, 1994]. There are three different methods for sensing electric fields: loading mode, shunt mode, and transmit mode. In loading mode, measurements are taken of the change in current lost at the transmitter. These changes in current are then used to analyze the position of the object being sensed in the electric field. In shunt mode, neither the transmitter nor the receiver is connected to the object. By measuring the change in capacitance between the transmitter and receiver, a grounded object within the field can be detected. Finally, in transmit mode, the transmitting electrode is in direct contact with the human body. Thus, the human body itself becomes the transmitter. The finger walking tracking device discussed in this document will be based on transmit mode electric field sensing.

4.1 Transmit Mode

In transmit mode, the transmitting electrode is attached directly to the object which is to be sensed. In the case of the finger walking interface, the electrode is either attached directly to the human body or to an alternate transmitter which fits over the hand, such as a glove. Receivers are then used to sense the changes in the electric field created by the transmitter. A simple circuit diagram of the process is shown in figure 3. The transmitter, *T*, is directly connected to the users hand and the user is free to move his hand and fingers around. The receiver, *R*, is set some distance, *D*, from the transmitter. This design allows for several capacitance's resulting from the electric field created by the transmitter to be

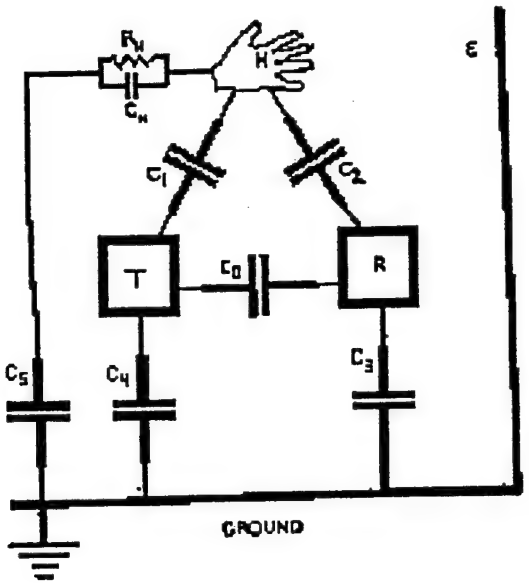


Figure 3: Lumped circuit model of Electric Field Sensing (Smith, 1996)

measured. The internal resistance and capacitance of the human body is represented by R_i and C_i respectively. The capacitance between the human body and ground is represented by C_4. The capacitance between the transmitter electrode and the receiver electrode is represented by C_2 and C_3 respectively. Finally, the capacitance between the human body and the receiver is represented by C_1. This capacitance, C_1, represents the geometry of the transmitter with respect to the receiver.

4.2 Capacitance Sensing

The capacitance sensed can be directly related to the distance, d, of the transmitting object from the receiver. The magnitude of the electric field falls off at a rate proportional to the inverse square of the distance, $1/d^2$, for point charges and at a linear rate, $1/d$, for

parallel plates. When the distance between the transmitting object (i.e. the user's fingers) and the receiving object (i.e. a wire) is small compared to the size of the receiving object, the two objects should be analyzed in terms of point charges. When this distance is small compared to the size, the objects should be analyzed as parallel plates. In the case of the finger walking interface, this distance is small when compared to the size of a typical user's fingers and the receiver required. Thus, the electric fields used in the finger walking interface are analyzed as parallel plates.

Measurements of the capacitance, C_l, sensed at the receiver can be used to determine the position of the transmitter with respect to the receiver. The measurements of the electric field consist of the superposition of the electric fields created by each component of the transmitter. Each charged element in a charge distribution contributes dE to the overall electric field surrounding the object, E. By integrating over the entire distribution of the field contributions from each charge in the distribution, the resultant electric field may be determined:

$$E = \int dE = \int \frac{dq}{4\pi\varepsilon_0 d^2} , \qquad (1)$$

where q is the charge of the transmitter and ε is the permittivity of the material the electric field is traveling through (ε_0 is the permittivity of free space). As can be seen from this equation, objects which are closer to the receiver contribute more to the magnitude of the electric field at the receiver than objects further from the receiver. By shielding the receiver from most of the human body and concentrating solely on the index finger and middle finger of the user, the measurements of the electric field will consist only of the superposition of the fields created by the two fingers.

When used to sense just the fingers of the user, the electric field will vary with changes in the position and geometry of each individual finger. The capacitance, C_j, is related to the charge, Q, of the object transmitting the electric field and the electric potential, V, sensed at the receiver:

$$C = \frac{Q}{V} \quad \text{or} \quad Q_i = \sum_j C_{ij} V_i \ . \tag{2}$$

By substituting in the strength of the field created for the charge, the above equation becomes

$$C = \frac{I_R}{2\pi f V} \ , \tag{3}$$

where I_R is the current of the receiver and f is the frequency of the field. This capacitance can be determined and used to analyze electric fields. The potential difference between two plates, neglecting fringing effects, is

$$V = \int_+^- E ds = \frac{q}{\varepsilon_0 A} \int_0^d ds = \frac{qd}{\varepsilon_0 A} \ , \tag{4}$$

where A is the area of the plates and d is the distance between the plates. Therefore, the capacitance between the plates, see figure 4, is

$$C = \frac{\varepsilon_0 A}{d} \ . \tag{5}$$

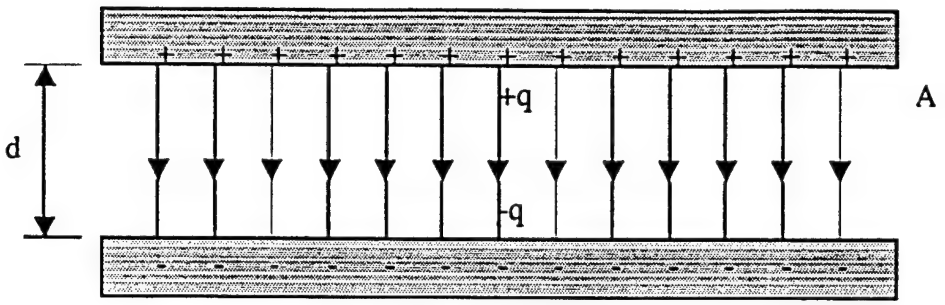


Figure 4: Cross section of a charged parallel plate capacitor.

The capacitance varies depending on both the geometry of and the distance between the two plates (receiver and transmitter). In other words, the distance between the transmitter and the receiver is

$$d = \frac{\varepsilon_0 A}{C}, \tag{6}$$

where the area, A, is a constant depending on the configuration of the receiver and transmitter and C is the capacitance measured by the electric field sensor.

4.3 Electric Field Proximity Sensor

The Electric Field Proximity Sensor (EFPS) developed at the MIT Media Lab incorporates this concept of electric field sensing into its design. The EFPS generates the electric field as a sinusoidal waveform which can be set between 10 and 115 kHz. The Electric Field Proximity Sensors output an electric potential (analog voltage) representing the field strength of the signal transmitted by the source at the receiver through synchronous detection which eliminates most of the electric-field noise associated with other nearby objects. The EFPS contains the circuitry required to create this electric field

and detect its strength at a specific location. The main advantage in using electric-field tracking is the lack of cumbersome equipment. The coupling of the user to the single transmitter can be accomplished through a simple button, which the user touches whenever he wishes to use the tracking device, or through a connection in the HMD. Since each EFPS costs only a few hundred dollars, the hardware for such a device is also relatively inexpensive.

The Electric Field Proximity Sensor board requires under 100 ma to produce and analyze the electric field. The circuitry of this board, specifically the receiver side filter and high gain operational amplifiers, and "a process called synchronous detection" [Bik, 1997], gives the system a high immunity to noise interference from local electric fields greater than 70 dB. For a complete schematic of the board and circuitry see Appendix A. This board can be operated in both the shunt mode and transmit mode. For the purposes of this study, the board will be used in the transmit mode only.

The transmitter of the finger walking interface is connected to the Electric Field Proximity Sensor board through a BNC connector. A frequency generator on the Proximity Sensor board creates a sinewave between 10-115 kHz. This frequency is specifically determined by the value of a potentiometer, T_2, which can be set on the board. The amplitude of this sinewave has a maximum value of 17 V peak to peak and is also set by a potentiometer, T_1.

The transmitted signal is received by an antenna connected directly to the Electric Field Proximity Sensor board through a BNC connector. The EFPS receiver can consist of any conductive material ranging from a single wire to a plate. An input filter compares the waveform of the signal detected with the signal sent and passes only the signal with the same waveform as the signal transmitter. The sensitivity of the receiver can be adjusted

through a potentiometer, T_4. The value of this potentiometer is an offset which is added to the signal received. The greater the offset, the more sensitive the proximity sensor board is to changes in the position of the transmitter.

Before being outputted from the proximity sensor board, the received signal passes through an amplification stage. The level of amplification and the range of the receiver signal is controlled though a final potentiometer, T_3. The output of this final stage ranges from 0 to +12 Volts.

When properly configured, the EFPS offers an inexpensive method of tracking the movement of a user's fingers across a fixed surface surrounded by an array of receivers. Table 1 shows the possible configurations of the EFPS. The number of boards required for a tracking system is dependent on the level of precision required by the system. For example, a two dimensional, single finger tracker requires one transmitter and at least two receivers, one for each direction (x, y). Thus, this type of system would need two proximity sensor boards.

Name	Type	Potentiometer Range	Effect
Output Amplitude	T1	0-100K ohm	0-17 peak-to-peak voltage
Frequency	T2	0-5K ohm	10-115 kHz
Range	T3	0-100K ohm	1-4 gain
Sensitivity	T4	0-100K ohm	0-5 Volts offset

Table 1: The four adjustable potentiometer settings on the Electric Field Proximity Sensor board and their effect on the system output.

5 Experiment #1—Tracking Performance

5.1 Objective

The aim of this experiment is to establish a set of criteria for the design of an EFPS system for tracking the movement of a user's fingers. Specifically, this experiment is designed to determine the affect on the efficiency and resolution of the EFPS system of certain factors: the size, type, and placement of the receiver; transmitter design; interference from the human body and outside sources; and the analog to digital conversion. In order to test these factors, different antennae arrays and transmitters are designed and tested.

Three key questions which should be answered are the size of the receiver, the type of receiver to be used (a length of wire, a conductive plate, etc.), and the placement of the receivers (in plane or out of plane with respect to the transmitters). A receiver with a large surface area has a greater operational range between the source transmitter and the receiver. In addition, the height of the user's fingers above the surface of the pad will affect the performance of the EFPS because the receiver detects the absolute distance between the transmitters and receivers, which depends on both the x and y components of the user's finger and the height of the user's finger from the surface of the pad. For example, a receiver in the plane of the transmitter will be more sensitive to changes in the height of the transmitter than receivers out of the plane. Thus, a wire mounted in the same plane as the surface of the pad will be more affected by changes in height than a plate mounted either vertically or at an angle to the surface of the pad.

The second design criteria is the type of material used for the transmitter and its design. The material will affect the magnitude of the electric field generated and, thus, the performance of the system. The greater the resistance of the material, the weaker the signal generated. Possible materials for the transmitters include aluminum, copper, and the human body. When the human body is used, an electrode either needs to be attached directly or capacitively coupled to the user's body. As with the performance of the EFPS receivers, the performance of the transmitters will be greatly affected by the surface area of the transmitter. Therefore, in this experiment, two transmitting electrodes will be used in place of a single electrode connected to the user's body. This simplification will aid in the development of the mechanical, electrical, and software systems required to create a finger walking interface.

The third major factor is the effect of conductive bodies in the electric field generated by the EFPS. Conductive bodies in electrical contact with the transmitter will begin broadcasting the electric field, thereby increasing the strength of the field detected by the receivers. On the other hand, grounded conductive bodies within the field will absorb the transmitted electric field, decreasing the strength of the field. Thus, the pad area should be as free as possible of other conductive bodies. One possible method to isolate the receiver from interference is to keep the human body electrically isolated from the transmitter (i.e. separate transmitters for each finger). Another possible method to eliminate outside interference is to construct a grounded shield around the pad which will absorb all interference from sources outside the direct surface of the pad (i.e. human body transmitting but only the fingers are detected).

The fourth design criteria is the method used to convert the analog electric potential out of the EFPS boards to a digital signal which can be manipulated by a computer. Specifically, there are three key factors of the analog to digital PC card chosen which may

affect system performance: resolution, speed, and error rate. Resolution is a measure of the precision of the analog to digital conversion. For example, a card with a 0.1 voltage resolution can detect changes in electric potential every 0.1 volts. The speed of an analog to digital card is a measure of the time required to acquire the data. If the card takes 0.1 seconds to acquire a sample, the fastest update rate for the finger walker system is 10 Hz, which is too slow for a virtual environment system. The error rate of the card is a measurement of the magnitude of the error in each conversion. For example, an error rate of 0.1 voltage means that every reading is accurate within 0.1 volts. All three of these factors should be minimized in order to maximize system performance.

The goal of this study is to compare different EFPS system configurations in order to determine which is the most effective. Four different receiver setups were designed to test the effects of receiver size: a four receiver array of 4" long 18 gauge wires, a four receiver array of 8" long 18 gauge wires, a four receiver array of 2" by 4" aluminum plates. Each of these setups are compared to each other through a number of experiments. This study will allow for a determination of the most effective receiver antennae design, as well as an assessment of the effectiveness of an EFPS system.

5.2 Experimental Apparatus

To test the four receiver designs and two transmitter designs, a test platform was designed and built. The construction of this test platform required three major system components: the mechanical system (section 5.2.1), the electrical system (section 5.2.2), and the computer software (section 5.2.3).

5.2.1 Mechanical System

The mechanical system designed for these receiver experiments incorporates three critical design features. First, the test platform requires a modular design which allows for components to be quickly switched in and out. With such a design, different receiver components can be tested with the same pad surface and transmitter. In addition, the test platform needs to be easily adapted to new concepts with regard to receiver design. A third factor is the need for a method to aid in the debugging of the EFPS system. Specifically, a grid on the surface of the walking interface is necessary to match specific finger positions to specific voltages.

A secondary concern in the design of the finger walking interface is the choice of material used in the construction of the pad. The walking surface requires a durable material which can withstand the abuse of constant use as a computer interface. The material must also have a low coefficient of friction, allowing user's fingers to slide across the interface surface, encountering only a slight amount of resistance. Thirdly, the material must also be easy to machine in order to manufacture the mounts required by the receivers and wires associated with the interface device. Finally, the material must be inexpensive in order to keep the price of the interface device as low as possible. Acrylic was chosen for the initial construction of the finger walking interface to meet all the material criteria (durability, low coefficient of friction, easily machineable, and low cost).

To meet these design criteria, a base plate was designed upon which each of the remaining system components (receivers, wires, and shielding) could be mounted. As can be seen by looking at the mechanical drawings of the base plate in figure 5, the design simply consists of a 8-1/2" by 8-1/2" square plate of acrylic with a grid for positioning and mounting holes for the receivers and shield. The base plate was designed to mount

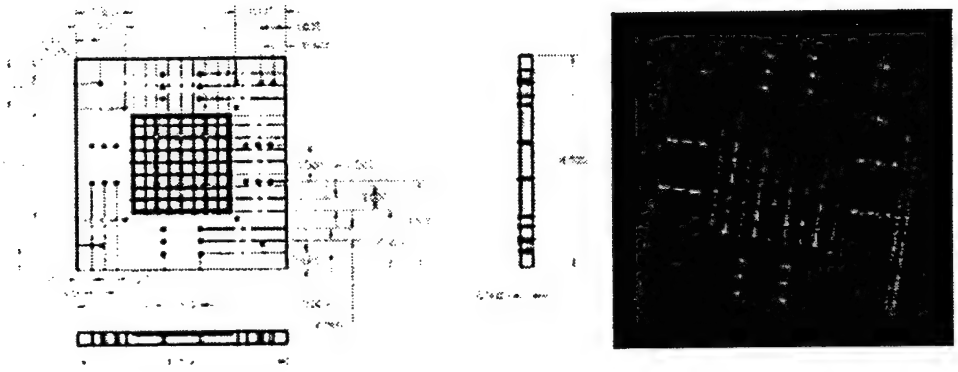


Figure 5: On the left side is the mechanical drawing of the base plate, while on the right is a picture of the actual acrylic plate.

brackets for the receivers at three separate distances offset from the edges of the walking surface. By measuring a test subject's finger walking range it was determined that a walking surface of approximately 4" by 4" would be more than adequate to allow a full range of motion for the user. Thus, a 4" by 4" section of the base plate acts as the walking surface and is overlaid with a grid. This positioning grid is milled 1/8" deep on the underside of the base plate. Because acrylic is clear, the grid can be seen through the plate while, at the same time, allowing the walking surface to remain smooth.

In order to meet the design criteria for modularity and adaptability, a system of brackets was designed to mount the receivers to the base plate. Several different brackets were designed to mount the different receivers. Each bracket consisted of three main parts: the receiver mount, the base plate mount, and the BNC cable mount. Figure 6 shows a typical receiver bracket used to mount a 4" long 18 gauge wire parallel to the surface of the base plate. It consists of a mounting for a BNC cable connection, the mounting grove for the wire, and the mounting holes for attaching the bracket to the base plate. The mounting groove allows the wire to be mounted slightly above the walking surface in order to test the effects of in plane and out of plane receivers.

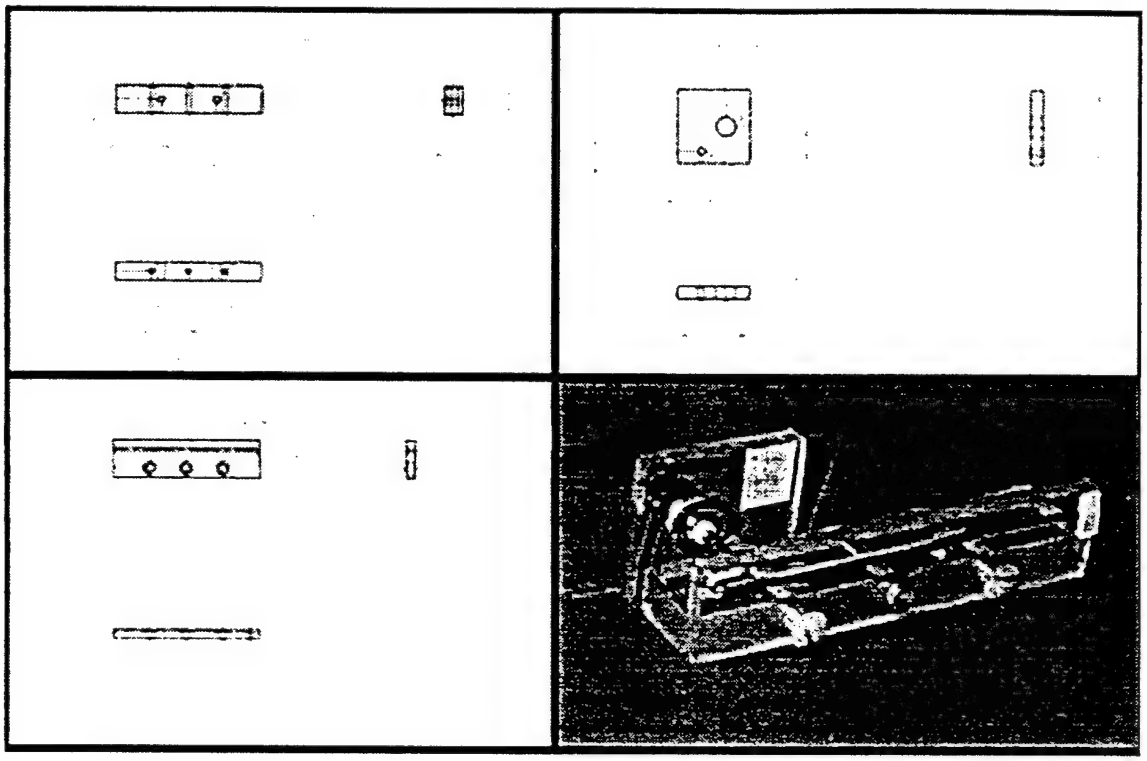


Figure 6: Receiver bracket for mounting a 4" long wire parallel to the surface of the base plate.

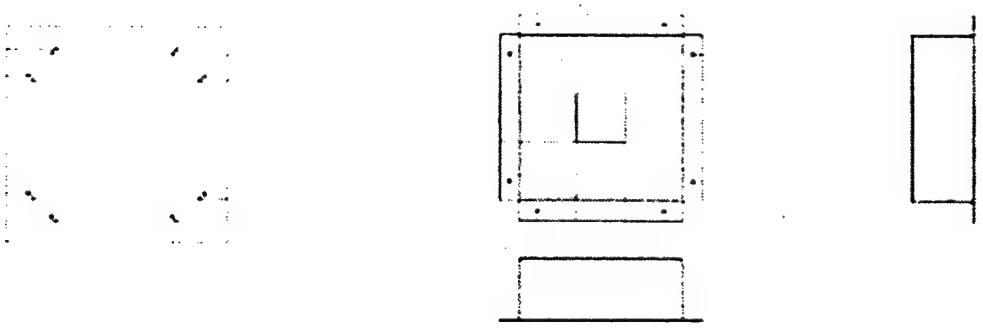


Figure 7: Mechanical Drawings of the ground plate and hood used to isolate the receiver array from outside interference.

In order to eliminate interference from outside sources (other conductive bodies, other unrelated electric fields, etc.), the base plate and receivers are surrounded by a grounded shield. The shield consists of two major parts which can be seen in figure 7: the bottom ground plate and the hood. The bottom ground plate consists of a 1/8" thick plate of aluminum with mounting holes for the hood and base plate. The base plate is elevated off the surface of the bottom ground plate by nylon standoffs. The hood slides snuggly over the base plate and receivers, isolating them from outside interference. While resting his wrist on the top of the hood, the user inserts his fingers through a 2-1/2" by 2-1/2" hole cut through the top section of the hood. In addition to this hole, one slit is cut in each of the four sides of the hood allowing the receivers to be connected to the EFPS boards.

5.2.2 Electrical System

The electrical system is crucial to the performance of the finger walking user interface. The EFPS is used to generate the electric field and analyze the electric potential at the receivers. Special modifications are required when using more than one receiver with a specific transmitter. An analog to digital converter is used to translate the analog voltage outputted from the EFPS's to a digital value which can be manipulated by a computer. Refer to Appendix C for the complete electrical schematics of this experimental system.

This system uses four Electric Field Proximity Sensor boards. In order to use more than one receiver to detect the same transmitted signal, the transmitted waveform must be available for each of the Electric Field Proximity Sensor boards. To do this, the signal from pin 2 of the ICL8038 precision waveform generator on the transmitting board must be attached by a wire to the .1 μf capacitor, C_4, on each additional board receiving the same

transmitted signal. In addition to this connection, the transmitter on the second board must be disabled by cutting the trace at pin 2 of the ICL8038 chip. See figure 8 for a schematic depicting this modification.

The boards are mounted in a case which also contains the system power supply. Both the connections to the pad and the connections to the analog to digital converter card are made through the rear panel of this case. The transmitters and receivers are connected through BNC cables. The receivers used in this experiment consist of either a single wire or an aluminum foil plate. The outputs of the EFPS cards are connected by a 9-pin connector to the analog to digital card.

The analog to digital conversion is done using a Keithley Data acquisition board, the DAS-1602. This card allows for 8 differential or 16 single-ended analog inputs at a resolution of 12-bits. The DAS-1602 can sample a ± 10 volt analog input 100 thousand times per second at selected gains (1, 2, 4, or 8) through any of three different modes: program control, interrupt service routine, and demand mode direct memory acquisition (DMA).

The DAS-1602 acquisition board meets all the necessary requirements of the finger walking user interface for the analog to digital conversion. The card has a resolution of 12 bits over a ± 10 volt range which allows the card to detect changes in voltage every 0.005 volts:

$$\text{Resolution} = \frac{V_{max} - V_{min}}{4096} = \frac{20}{4096} \approx .005V \ . \qquad (7)$$

The maximum delay of a DAS-1602 conversion is 8.5 us which sets an upper bound on system speed at approximately 100 kHz:

$$\text{Max frequency} = \frac{1}{time} = \frac{1}{8.5\mu s} \approx 100KHz\,. \qquad (8)$$

Finally, the error rate of the analog to digital card is 0.02% of the data reading within ± one least significant bit. In other words, the maximum error of any reading will be ± 7 milivolts:

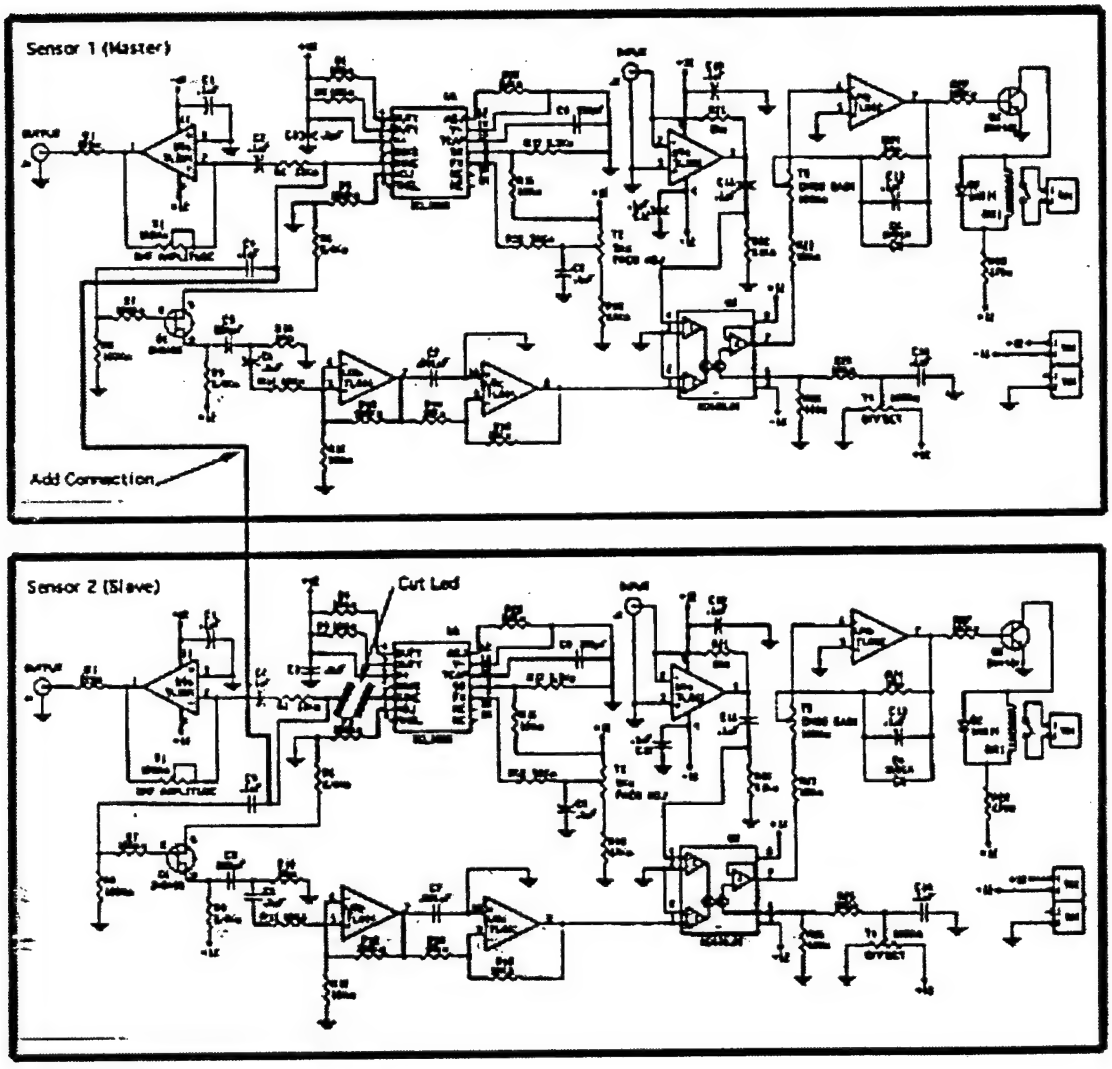


Figure 8: Modifications to the Electric Field Proximity sensor required to setup a system consisting of one transmitter and two receivers.

$$\begin{aligned} \text{Max Error} &= \pm\left[(V_{max})\cdot(0.0002)+(1LSB)\right] \\ &= \pm\left[(10V)\cdot(0.0002)+(0.005V)\right]. \\ &= \pm 0.007V \end{aligned} \tag{9}$$

The system software provided with the analog to digital card is used to configure the card. Using this software, D1600.CFG, the card is set to 10 MHz clock which results in a sample rate of approximately 100 kHz and 8 differential bipolar inputs. The required configuration files are available in Appendix D.

The main analog and digital input/output connector for the DAS-1602 is a 37-pin D-type connector accessed at the rear of the computer. A schematic of the connection between the Electric Field Proximity Sensors and the analog to digital card is shown in figure 9.

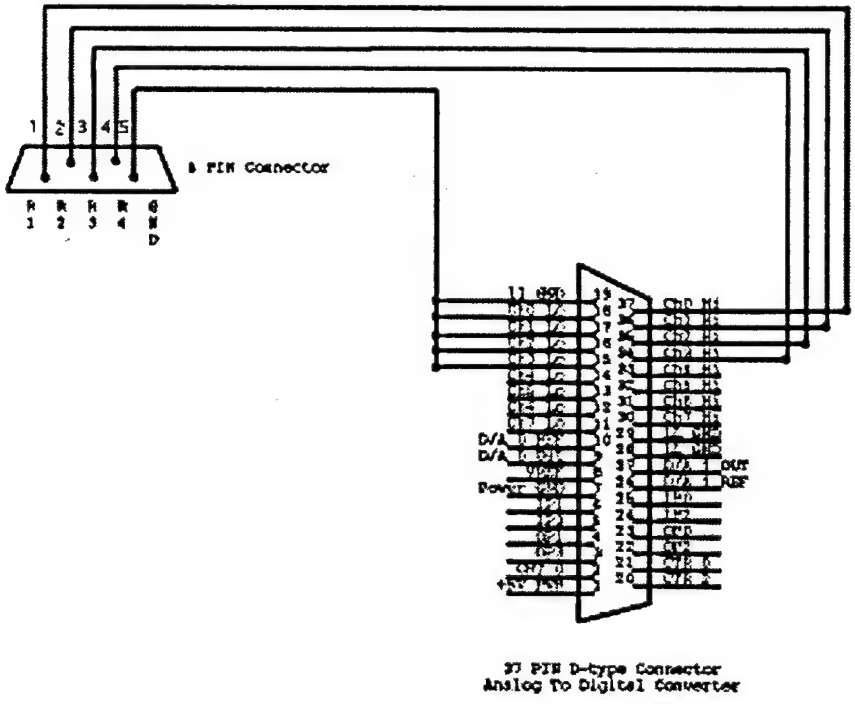


Figure 9: Connection between the EFPS and the analog to digital card.

The ground plate and hood are electrically grounded through the ground wire associated with the BNC coax cable for receiver 1. A wire from the BNC connector on receiver 1 is attached to the ground plate.

As previously mentioned, the key factors affecting the performance of the transmitter design are the surface area and the resistance of the transmitter. To test these factors, two types of transmitters were designed: a glove design and a thimble design. The glove consists of a ring of charge around the tip of the finger of both the fore and the middle fingers of a thin glove. Each ring of charge is connected to a separate transmitter. The thimble design consists of two sewing thimbles also connected to separate transmitters. Both designs were initially tested to determine their effect on the performance of the system

5.2.3 Computer Software

The software provided with the analog to digital card includes a control panel for viewing each of the channels from the analog to digital card. This application is used to quickly analyze the output of the EFPS boards to determine if the four potentiometers on each board are properly adjusted. The adjustments are made so as to maximize the distance traveled by the user's finger as it moves away from the receiver and the electric potential decreases from 10 volts to 0 volts. Once properly adjusted, the EFPS will output a unique set of electric potential readings depending on the position of the transmitter over the pad.

The Keithley Data Acquisition ASO-1600/1400 Advanced Software Package is used to control the DAS-1602 card. This package supports command calls from C, Pascal, and Dynamic Link Libraries (DLLs). Presently, the Finger Walker uses the C++ functions provided with the 32 bit software library Dasshl32.lib for initializing a connection

(*K_OpenDriver*), communicating with the drivers (*K_GetDevHandle*), retrieving a sample from the card (*K_ADRead*), and closing a connection (*K_CloseDriver*). See appendix E for details on these function calls. The C++ code for acquiring one data sample from each of the four channels used on the analog to digital card is shown in figure 10. First, the hardware and software are initialized. Next, communication is established with the driver through a device handle. Then the data samples are acquired from the relevant channels of the analog to digital card. Finally, the driver is closed and all relevant resources are released.

```
{
      DWORD hDrv1600;                  // Driver Handel
      DWORD hDev1600;                  // Device Handle
      short nErr;                      // Function return error flag
      WORD wADval;                     // Storage for A/D value

// Initialize the hardware and software
if(( nErr = K_OpenDriver( "DAS1600", "das1600.CFG", &hDrv1600)) != 0)
{
putchar(7);
printf("Error %X during K_OpenDriver", nErr);
exit(nErr);
}

// Establish communication with the driver through a device handle
if(( nErr = K_GetDevHandle(hDrv1600, 0, &hDev1600)) != 0)
{
putchar(7);
printf("Error %X during K_GetDevHandle ", nErr);
exit(1);
      }

      // Read channel 0 at gain 1; stor sample in Advalue
if((nErr = K_ADRead(hDev1600, 0, 0, &wADval)) != 0)
{
putchar(7);
printf("Error %X in K_ADRead operation.", nErr);
exit(1);
}

// Strip channel tag and display ADvalue
printf("A/D value from channel 0 is: %x\n", (wADval>>4)&0xfff);

// Close the Driver and Release All Resources
K_CloseDriver(hDrv1600);

return 0;
}
```

Figure 10: Analog to digital code to acquire a sample from channel 0 at a gain of 1 and print the data to the screen.

The data acquired from the analog to digital card by the *K_ADRead* function is stored in the form of counts. The analog to digital card acquires the electric potential reading by charging a capacitor and storing the time required for the capacitor to discharge. The analog to digital driver returns values left-justified in the lower 16 bits of a long integer. To unpack this data into a variable which represents the counts, the data in the long integer must be stripped of the channel tag and right shifted four places:

$$\begin{aligned} &\text{Actual Value} = (\text{right - shift data 4 places}) \text{ bit - wise AND with 0fff} \\ &\text{Counts[a]} = (\text{wADval} >> 4) \,\&\, 0\text{xfff} \qquad (\text{C++ Code}) \end{aligned} \tag{10}$$

The C++ code for stripping the channel tag and shifting the data is shown in the second half of equation 10. This variable can then be translated into its voltage equivalent through the following equation:

$$\begin{aligned} &\text{Voltage} = \frac{(count - 2048) \times (V_{\max} - V_{\min})}{4096} \qquad (\text{Bipolor}) \\ &\text{Voltage} = \frac{Counts \times V_{\max}}{4096} \qquad (\text{Unipolor}) \\ &\text{Voltage} = \left(\left(\left((double)Counts[a]\right) - 2048\right) * 20\right) / 4096; \qquad (\text{C++ Code}) \end{aligned} \tag{11}$$

where V_{max} is the maximum voltage read by the analog to digital card (+10 Volts in this experiment) and V_{min} is the minimum voltage read by the analog to digital card (-10 Volts in this experiment).

To evaluate the different receiver arrays, data must be collected and stored in a text file. Microsoft Excel contains a feature which allows tab delimited files to be opened within the spreadsheet program. Thus, by storing the voltage reading from each channel of

RN X Y SN V_1 V_2 V_3 V_4 C_1 C_2 C_3 C_4

Figure 11: Tab delimited document format.

the analog to digital card in a tab delimited text file, the receiver arrays can be analyzed and compared to one another. The tab delimited document format used for this experiment is shown in figure 11. The first column signifies the position number of the sample, or *SampleNumber* (*SN*). Each position on the grid has a *SN* associated with it. For example, *SN*=1 . The second and third column signify the x and y coordinates of the sample. The fourth column indicates the *RunNumber* (*RN*) of the data at that position. For example, the first data sample at position (1, 2) will have *RN*=1 and the second will have RN=2. The next four columns signify the electric potentials (V_1, V_2, V_3, V_4) acquired by the analog to digital card from each of the four EFPS boards which is determined by applying equation 11 to the counts (C_1, C_2, C_3, C_4) collected from the analog to digital card. The code to output the data acquired by the analog to digital card to this tab delimited text format is shown in figure 12.

Using the functions for capturing data from the analog to digital card and outputting to a tab delimited file, five different programs were written to test the various properties

```
int SampleNumber=1;          // Sample position number (1-64)
int PosX=0;                  // X-coordinate on grid
int PosY=0;                  // Y-coordinate on grid
double Voltage[4];           // Converted Voltage reading
int Counts[4];               // Counts reading from a2d card
int RunNumber=1;        // Run Number of data at a single point (1-10)
int output_text(void)
```

Figure 12: Tab delimited text output C++ code.

associated with the EFPS, the transmitter, the receiver, and the analog to digital card. These five programs can be seen in Appendix E. *Tbed0.c* collects ten data points every inch along x, y plane (25 points). *Tbed1.c* collects ten data points every 1/2" along x, y plane (81 points). *Tbed2.c* collects ten data points every 1/2" along x, y plane (81 points) with a one second delay between samples. *Tbed3.c* collects one data point every 1/2" along x, y plane (81 points) repeated ten times. Finally, *Height.c* collects ten data points every inch along x, y plane (25 points) at five different elevations (0", 1/4", 1/2", 3/4", and 1").

5.3 Experimental Methods

Since the type and placement of the receivers is crucial to the operation of the finger walker, a study of the resolution (section 5.3.3) of three different receiver setups (section 5.3.2) was performed in order to determine a suitable configuration. In addition, each time the finger walker interface is reconfigured with a new receiver or transmitter, the EFPS must be adjusted (section 5.3.1).

5.3.1 Adjusting EFPS

Adjusting the settings on the EFPS board is crucial to the performance of the finger walker interface. The EFPS adjustments can be split into two phases, both using the ASO-1600 control panel. Each time the transmitter or receiver is replaced, the EFPS settings must be readjusted.

The first phase in adjusting the EFPS is the setting of the frequency and amplitude. The amplitude should be set to maximum at all times (T_1 Full Clockwise Position). In order to determine the frequency for each transmitter and receiver pair, the proper phase and wavelength must be found through trial and error. With the range (T_3 Full Counterclockwise) and the sensitivity (T_4 Full Counterclockwise) set to their minimum values, the frequency is slowly varied until the voltage received is maximized.

Once the frequency and amplitude are set, the range and sensitivity can be adjusted to maximize the performance of the selected receiver and transmitter pair. Ideally, when adjusting receiver 1, the output voltage displayed on the ASO-1600 control panel with the transmitter at the points on the grid furthest from the receiver (transmitter located at positions (4, 4) and (4, 0)) should be 0 volts, and the reading with the transmitter at the point closest to the receiver (transmitter located at position (0, 2)) should be 10 volts. In order to maximize the range covered by the EFPS system, the range should be set as low as possible and the sensitivity should be adjusted to shift the voltages to the correct position.

5.3.2 Receiver Designs

The three receiver setups are discussed in the following paragraphs. The three receiver arrangements, as seen in figure 13, are discussed in the following paragraphs.

Receiver 1

The first receiver setup consists of four, four-inch 18 gauge wires mounted along the edges of a four-inch by four-inch grid in plane with the surface of the pad (see figure 13a). The wire is elevated off the surface of the pad in order to receive the transmitted electric field in

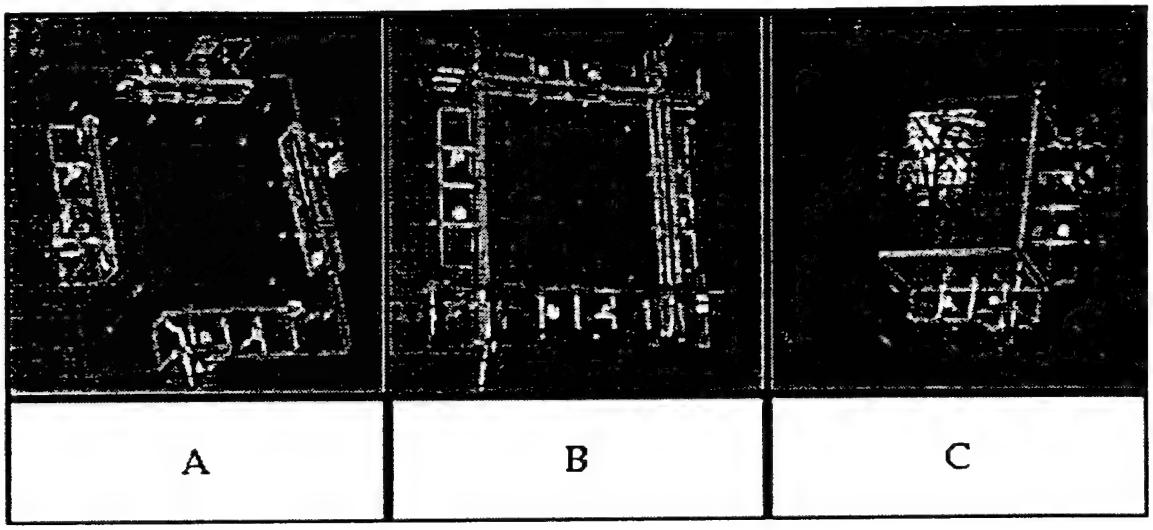


Figure 13: The three different receiver configurations. (a) Shows Receiver 1 at a 3" offset. (b) Shows Receiver 2 at a 1-1/2" offset. (c) Shows Receiver 3 at a 0" offset.

the same plane as the moving finger. These receiver brackets can be offset from the edge of the walking surface by any of the three distances (0", 1-1/2", and 3").

Receiver 2

The second setup consists of four, eight-inch 18 gauge wires placed along the edges of a four-inch by four-inch grid in the plane of the pad (see figure 13b). As with receiver 1, the wire in receiver 2 is elevated off the surface of the pad in order to receive the transmitted electric field in the same plane as the moving finger. These receiver brackets can be offset from the edge of the walking surface at only one of the three offset distances (1-1/2").

Receiver 3

The third setup consists of four, two-inch by four-inch conductive plates placed perpendicularly along the edges of a four-inch by four-inch grid out of plane with the surface of the pad (see figure 13c). Unlike receivers 1 and 2, the plate is not in plane with the moving finger but, instead, is out of plane with the moving finger in order to decrease

the effect of changes in elevation in the transmitter. These receiver brackets can be offset from the edge of the walking surface by any of the three distances (0", 1-1/2", and 3").

5.3.3 Experiments

For this study, each of the three setups described above uses two receivers to detect each transmitter, one along the x-axis and the other along the y-axis. Four experiments were conducted in order to test the performance of the receiver arrays described previously in section 5.3.2: testing initial settings of the EFPS system, testing the resolution of the system, testing the repeatability of the system, and testing the effect of varying the height of the receiver over the surface of the pad. These experiments are described in the following paragraphs (see also Appendix F for the code for the test programs).

Initial Settings

First, the initial settings of the EFPS are tested to ensure a complete range of voltages across the entire surface of the pad. Both the ASO-1600 control panel and the *Tbed0.c* test program are used in this experiment. The ASO-1600 control panel is used as described in section 5.3.1 to adjust the EFPS for each receiver array. Once adjusted, the *Tbed0.c* program is used to capture ten electric potential samples at twenty-five different positions (one position every 1" on the grid) and load it into Excel for analysis. During the test, the transmitter is placed at the first test position (0", 0") and ten electric potential samples are acquired. The transmitter is then moved to the next position (1", 0"). This process continues until the tenth sample is acquired at the twenty-fifth position (4", 4"). All three receiver arrays, both transmitters, and varying levels of shielding were tested in multiple configurations. The data was then stored in text files for later analysis in Excel (section 5.4.1).

Resolution

Next, the resolution of the EFPS and receiver arrays is tested to determine the factors affecting receiver performance. Both the *Tbed1.c* and *tbed2.c* test programs are used in this experiment. The *Tbed1.c* program is used to capture the data at 81 different positions separated by 1/2" and load it into Excel for analysis of the resolution of the receiver setup. The *Tbed2.c* program is used to capture ten data samples separated by one second at 81 different positions separated by 1/2" in order to test the variance of the setup. For both these tests, the transmitter is placed at the first position (0", 0"), ten samples are acquired, the transmitter is moved to the next position (0", 1/2"), etc. All three receiver arrays, both transmitters, and varying levels of shielding were tested in multiple configurations. The data was then stored in text files for later analysis in Excel (section 5.4.2).

Repeatability

Thirdly, the repeatability of the EFPS and receiver arrays is tested to determine the factors affecting receiver performance. The *Tbed3.c* test program is used in this experiment to capture the data at 81 different positions separated by 1/2", ten times in a row, and load it into Excel for analysis of the repeatability of the receiver setup. Unlike the two previous experiments, this test acquires a single sample at the first transmitter position (0", 0"). The transmitter is then moved to the next position (0", 1/2") and, again, only one sample is acquired. After a single sample is acquired at each of the 81 positions the process begins again at the original point (0", 0"). This process is repeated until ten samples have been acquired at each of the 81 positions on the grid. All three receiver arrays, both transmitters, and varying levels of shielding were tested in multiple configurations. The data was then stored in text files for later analysis in Excel (section 5.4.3).

Height

Finally, the influence of different transmitter heights above the surface of the pad is tested to determine the factors affecting receiver performance. The *Height.c* test program is used in this experiment to capture data at 25 different positions separated by 1" for each of four elevations, and load this data into Excel for analysis of different transmitter heights over the pad for each receiver setup. As with the first experiment, this experiment acquires ten samples are taken consecutively at each point. All three receiver arrays, both transmitters, and varying levels of shielding were tested in multiple configurations. The data was then stored in text files for later analysis in Excel (section 5.4.4).

5.4 Analysis/observations

The data from the three test experiments are analyzed within Microsoft Excel: initial settings (section 5.4.1), resolution (section 5.4.2), and repeatability (section 5.4.3).

5.4.1 Initial Settings

The initial settings experiment proved extremely useful both in analyzing the settings of the EFPS boards and eliminating several of the receiver setups and one of the transmitter designs. Once the EFPS settings were fixed using the ASO control panel, the voltage with respect to position was plotted within Excel. A sample plot of receiver 1 detecting the thimble transmitters at 1-1/2" offset is shown in figure 14. The plot shows lines of constant voltage detected by one of the four receivers. The goal of this experiment is to set the EFPS tracking system for each setup to maximize the distance traveled by the transmitter, up to 4", as the output of the EFPS board falls from 10 volts to 0 volts. Only four of the receiver setups achieved this range. Some of the receiver setups could only be

set to detect 2" of movement. Table 2 lists each of the possible receiver configurations for the thimble transmitters with the effective range of the system. The glove transmitter was quickly eliminated as a possible receiver design because too much of the signal was being absorbed by the human hand. In addition to eliminating several of the designs, the initial settings experiments also demonstrated the need for the grounded shield and grounded human body. The grounding greatly reduced the deviation between samples taken at identical positions.

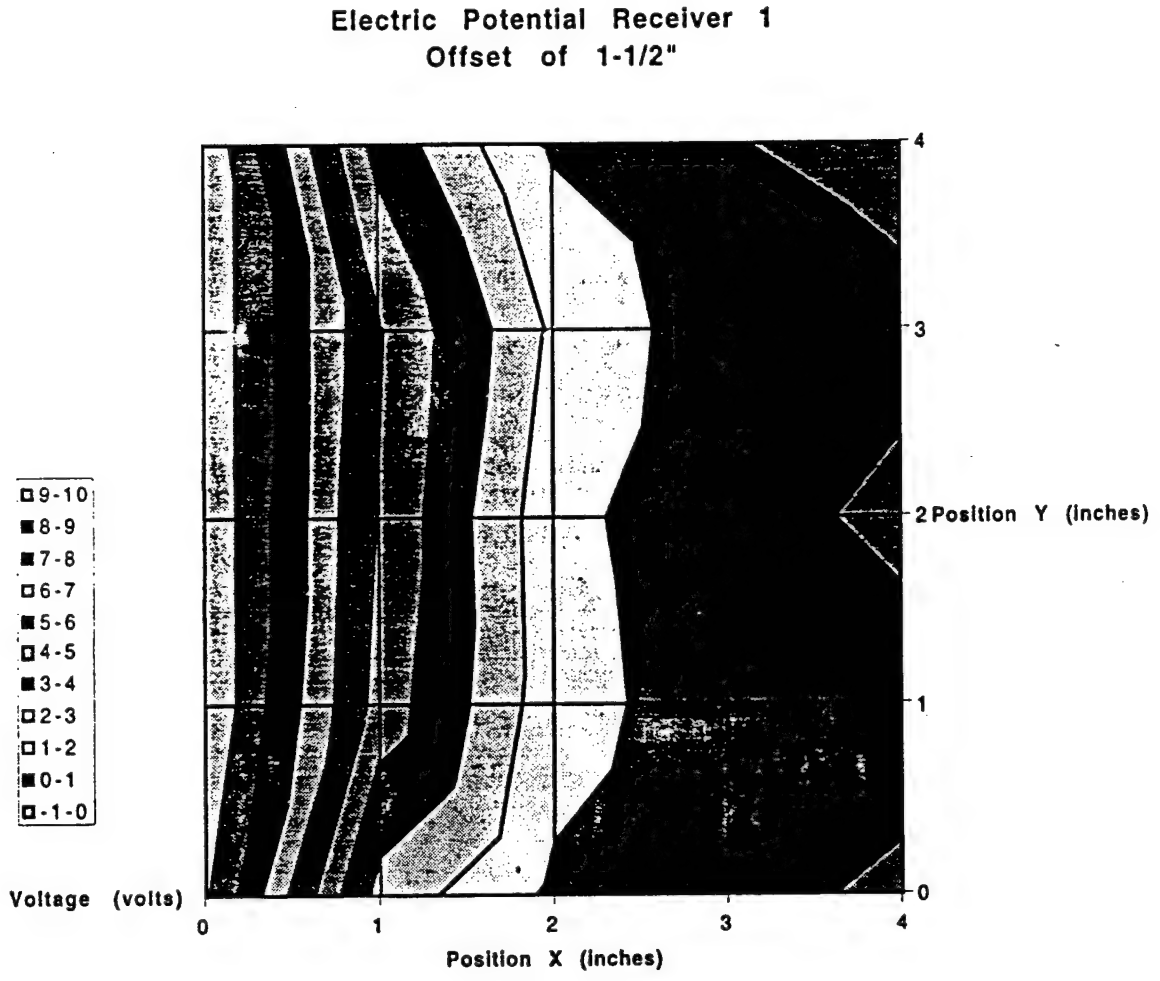


Figure 14: Sample initial settings plot of the electric potential detected by a type 1 receiver at a 1-1/2" offset.

Receiver Number	Offset	Distance Covered from 0-10 Volts	Eliminated?
1	0"	4"	No
1	1-1/2"	4"	No
1	3"	2"	**Yes**
2	0"	3"	No
2	1-1/2"	4"	No
2	3"	3-3/4"	No
3	0"	1-1/2"	**Yes**
3	1-1/2"	3-1/4"	No
3	3"	2"	**Yes**

Table 2: Results of initial tests on the different receiver configurations.

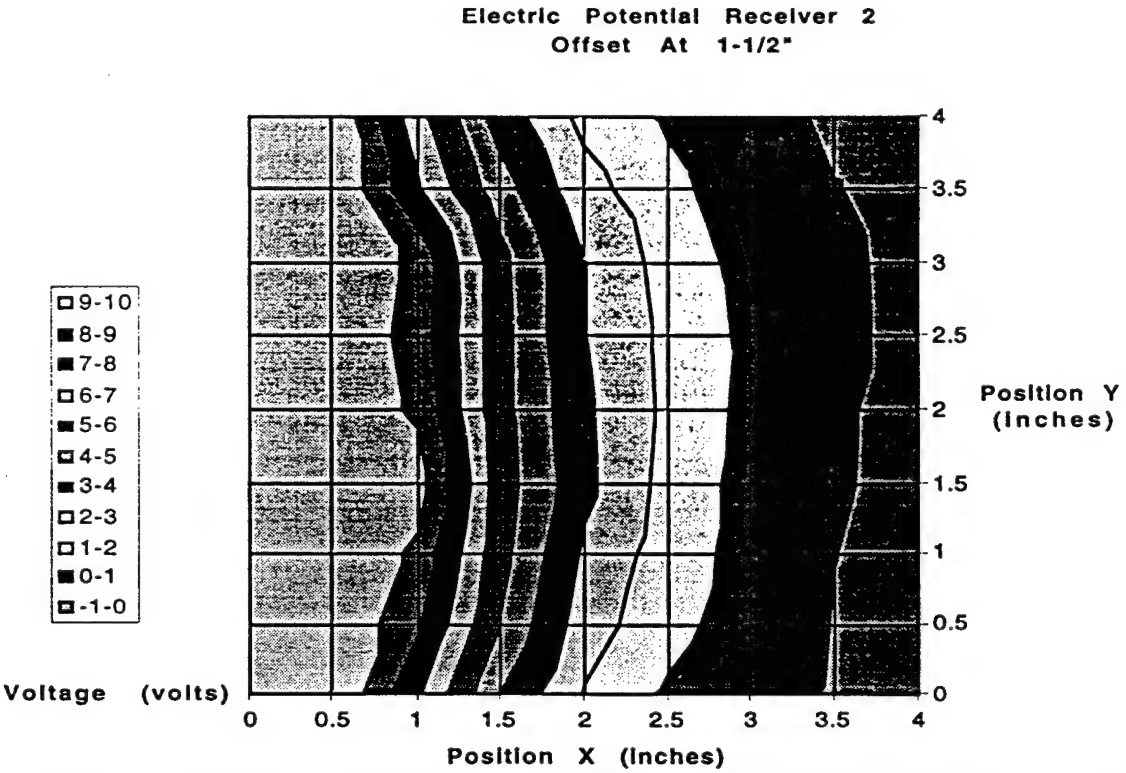


Figure 15: Sample resolution plot of the electric potential detected by a type 2 receiver at a 1-1/2" offset.

5.4.2 Resolution

The critical factor affecting system performance is the resolution of the system. To the best extent possible, each finger position must result in a unique electric potential. The resolution of each receiver setup was determined by plotting the voltage with respect to position as in section 5.4.1. The resolution plot for receiver 2 with a 1-1/2" offset is shown in figure 15. This plot was created by calculating the mean of the ten electric potential samples at each of the twenty-five positions. The plot shows the lines of constant voltage at every volt (0-10 volts) detected by the receiver. As can be seen in the figure, the lines of constant voltage arc slightly towards the ends of the receiver. Also, the change from a linear rate of change in the voltage to that of the inverse square can be seen in the plot at approximately x=2". Receivers 2 and 3 at a 1-1/2" offset minimize this curvature allowing for a constant voltage to be closely associated with the same point.

5.4.3 Repeatability

The repeatability of the receiver system is crucial to the overall performance of the finger walker interface. If the electric potential is not identical each time the transmitter is placed on a specific point on the grid, the EFPS interface will not be reliable as an input device for a virtual environment. Therefore, the mean and standard deviation of the ten samples for each position on the pad was calculated and plotted. Figure 16 shows the calculated mean and standard deviation for receiver 3 at a 1-1/2" offset. This plot is a slice of the standard deviation plot, with the y position of the finger held constant at 2" and the x position varied between 0" and 4". The measured voltage at every 1/2" is displayed with the standard deviation at each position represented by vertical bars above and below this point. Most of this deviation can be attributed to the difficulty of returning the finger to the

exact point at which the data was previously collected. As can been seen in the figure, the standard deviation, though small, is a function of the distance from the receiver. The points further from the receiver have a much smaller deviation than those near the receiver because the electric potential further from the receiver is decreasing at a rate proportional to the inverse square of the distance. Slight changes in position of the transmitter near to the receiver have a greater affect on the magnitude detected than movements further from the receiver. None of the systems had significant standard deviations at any point along the surface of the pad.

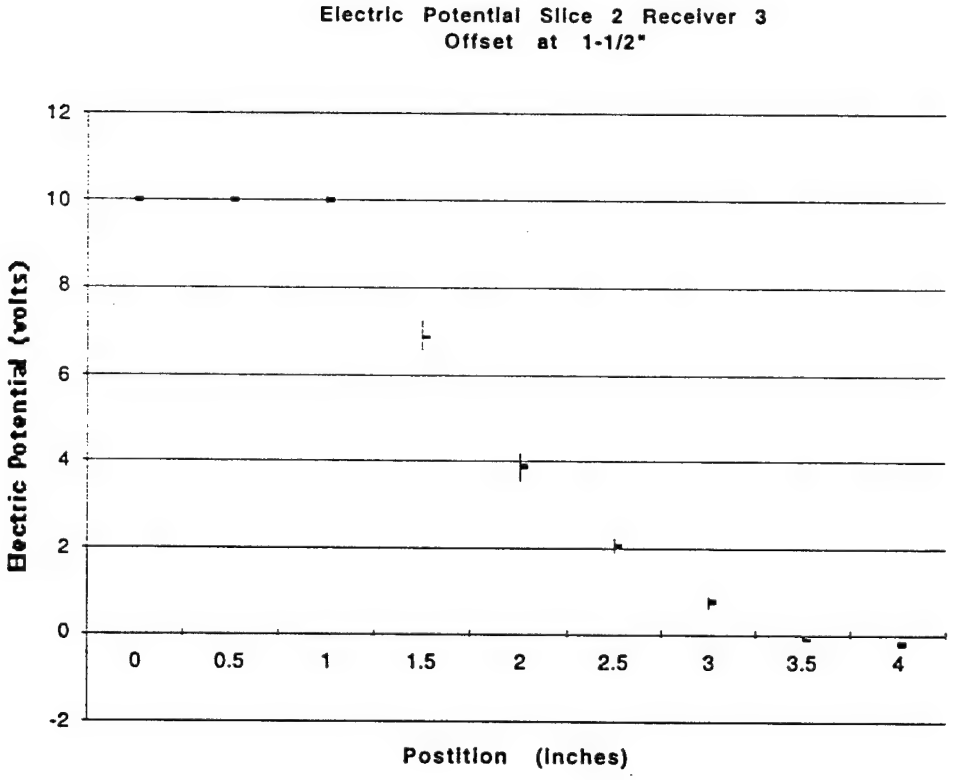


Figure 16: Excel plot of the mean and standard deviation for a type 3 receiver at a 1-1/2" offset. For this plot, the y coordinate of the user's finger is held constant at 2", while the x coordinate varies from 0" to 4".

5.4.4 Height

The height experiment proved that the level of the transmitter above the surface of the pad was not crucial to the performance of the finger walker. The output of the EFPS for the wire receivers was mildly effected by different heights, while the plate setups were virtually unaffected by changes in finger height. The main reason for this is the close proximity between the transmitter and the receiver. The effects of varying the height is only evident at the outer reaches of the pads usable space. If a wire is actually used in the full-scale walker, the effects of changes in height on the EFPS output will most likely become noticeable.

5.5 Results

After careful review of the analysis of the data in section 5.4, the array of four 8" long wires at an offset of 1-1/2" was chosen as the receiver setup for the finger walker interface. When compared to the other setups, this receiver array provided the best resolution for the system. In addition, the change in electric potential detected by the 8" receiver between positions on the pad was the closest to a linear relationship.

6 Methods to Determine the Position Vector

The electric potential from the output of the Electric Field Proximity Sensor must be translated into two position vectors. One vector represents the position of the fore finger of the user and the other vector represents the position of the middle finger of the user. In a computer program, the electric potentials can be processed using two different methods: mathematically in real time (section 6.1) or in a lookup table (section 6.2). Both methods have advantages and disadvantages which affect the overall performance of the finger walker user interface, as discussed in section 6.3.

6.1 Real Time Processing

The most accurate method for determining the position of the user's fingers is though the direct solution to Maxwell's equations (which will not be discussed in detail in this document) uses Maxwell's equations. For an in-depth review of this method for analyzing electric fields see Smith, 1995. A brief summary of the results using Maxwell's equations of the reference follows

$$\begin{aligned} \nabla \times E_1 &= \mu \frac{\partial H_0}{\partial t} \\ \nabla \times H_1 &= \varepsilon \frac{\partial E_0}{\partial t} + J_{f1} \\ \nabla \cdot \varepsilon E_1 &= \rho_{f1} \\ \nabla \cdot \mu H_1 &= 0 \\ \nabla \cdot J_{f1} &= \frac{\partial \rho_{f1}}{\partial t} \end{aligned} \qquad (12)$$

The Laplace equation is required in order to solve the Maxwell equations above. Laplace's equation for an inhomogeneous material is

$$-\nabla \cdot \varepsilon E_0 = \nabla \cdot (\varepsilon \nabla \phi_0) = \varepsilon \nabla^2 \phi_0 + \nabla \varepsilon \cdot \nabla \phi_0 = 0 \,. \tag{13}$$

As in section 4.2, the capacitance between two conductive bodies can be determined.

$$Q_i = \sum_j C_{ij} V_j \tag{14}$$

By Solving Laplace's equation and the capacitance equations, the distance of an object from the receiver can be determined.

An approximation of Maxwell's equations results in the electrostatic equations discussed in section 4. The distance, d_i, between the transmitter, T_i, and the receiver, R_i, is

$$d_i = \frac{\varepsilon_0 A_i}{C_i} \,. \tag{15}$$

Once the distance is determined between each of the transmitter and receiver pairs, the position of the transmitter with respect to the receivers can be determined through triangulation.

During operations (see figure 17), the EFPS will output the electric potential measured between the transmitter, T_i, and each receiver, R_i, to the computer through the analog to digital card. These electric potentials will be translated to a distance using

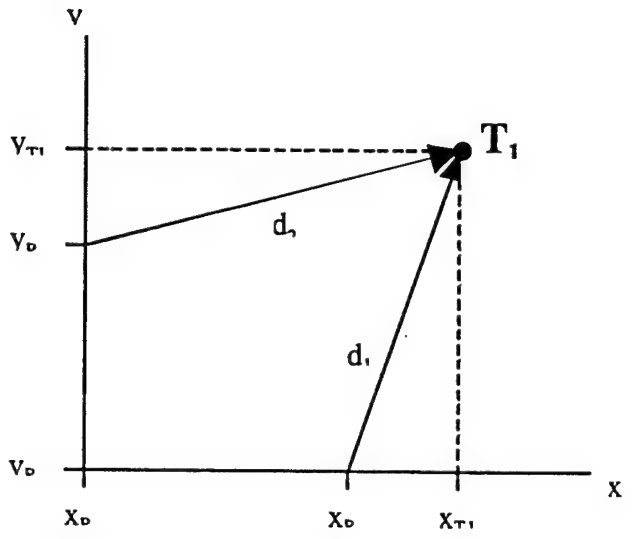


Figure 17: Normal operation, triangulation.

equation 12. The center position of each of the two receivers are fixed at the *points* R_1 (x_{r1}, y_{r1}) and R_2 (x_{r2}, y_{r2}). The transmitter, T_1, is located at some point (x_{t1}, y_{t1}) within the first quadrant of the x, y plane created by the placement of the two receivers. Using triangulation, the distances, d_1 and d_2, from the receivers to the transmitter determine the position of the transmitter.

$$\begin{aligned} \left(x_{T_1} - x_{R_1}\right)^2 + \left(y_{T_1} - y_{R_1}\right)^2 &= d_1 \\ \left(x_{T_1} - x_{R_2}\right)^2 + \left(y_{T_1} - y_{R_2}\right)^2 &= d_2 \end{aligned} \quad . \tag{16}$$

Thus, the current position of the transmitter, T_1, (x_{t1}, y_{t1}) is determined. The position of the second transmitter, T_2, (x_{t2}, y_{t2}) is found in a similar manner.

Voltage Receiver 2 \ Voltage Receiver 1	0	1	2	3	4	5	6	7	8	9	10
0	(4, 4)	(4, 3.5)	(4, 3)	(4, 2.5)	(4, 2)	(4, 1.5)	(4, 1.25)	(4, 1.25)	(4, 1)	(4, 0.5)	(4, 0)
1	(3.5, 4)	(3.5, 3.5)	(3.5, 3)	(3.5, 2.5)	(3.5, 2)	(3.5, 1.5)	(3.5, 1.25)	(3.5, 1.25)	(3.5, 1)	(3.5, 0.5)	(3.5, 0)
2	(3, 4)	(3, 3.5)	(3, 3)	(3, 2.5)	(3, 2)	(3, 1.5)	(3, 1.25)	(3, 1.25)	(3, 1)	(3, 0.5)	(3, 0)
3	(2.5, 4)	(2.5, 3.5)	(2.5, 3)	(2.5, 2.5)	(2.5, 2)	(2.5, 1.5)	(2.5, 1.25)	(2.5, 1.25)	(2.5, 1)	(2.5, 0.5)	(2.5, 0)
4	(2, 4)	(2, 3.5)	(2, 3)	(2, 2.5)	(2, 2)	(2, 1.5)	(2, 1.25)	(2, 1.25)	(2, 1)	(2, 0.5)	(2, 0)
5	(1.5, 4)	(1.5, 3.5)	(1.5, 3)	(1.5, 2.5)	(1.5, 2)	(1.5, 1.5)	(1.5, 1.25)	(1.5, 1.25)	(1.5, 1)	(1.5, 0.5)	(1.5, 0)
6	(1.25, 4)	(1.25, 3.5)	(1.25, 3)	(1.25, 2.5)	(1.25, 2)	(1.25, 1.5)	(1.25, 1.25)	(1.25, 1.25)	(1.25, 1)	(1.25, 0.5)	(1.25, 0)
7	(1.25, 4)	(1.25, 3.5)	(1.25, 3)	(1.25, 2.5)	(1.25, 2)	(1.25, 1.5)	(1.25, 1.25)	(1.25, 1.25)	(1.25, 1)	(1.25, 0.5)	(1.25, 0)
8	(1, 4)	(1, 3.5)	(1, 3)	(1, 2.5)	(1, 2)	(1, 1.5)	(1, 1.25)	(1, 1.25)	(1, 1)	(1, 0.5)	(1, 0)
9	(0.5, 4)	(0.5, 3.5)	(0.5, 3)	(0.5, 2.5)	(0.5, 2)	(0.5, 1.5)	(0.5, 1.25)	(0.5, 1.25)	(0.5, 1)	(0.5, 0.5)	(0.5, 0)
10	(0, 4)	(0, 3.5)	(0, 3)	(0, 2.5)	(0, 2)	(0, 1.5)	(0, 1.25)	(0, 1.25)	(0, 1)	(0, 0.5)	(0, 0)

Figure 18: Two dimension, eleven by eleven, lookup table with inputs V_1 and V_2.

6.2 Preprocessing in a Lookup Table

Because processing the position vector for each finger is extremely demanding in terms of computational power in a PC, a lookup table can be used in place of the equations. The position vectors are preprocessed and stored in the computer's memory for future access. These memory locations correspond to the voltages received by the analog to digital card. For example, when two transmitters are used, the voltages received by receivers 1 and 2 are used to access the position of transmitter 1 (see figure 18). Similarly, the voltages of receivers 3 and 4 are used to access the correct memory location containing the position of the second transmitter. The previous example uses two, two dimensional lookup tables to determine the position vectors of the transmitters. The C++ code for initializing and accessing such lookup tables is shown in figure 19. Both the number of dimensions and the size of the lookup table can be varied in order to affect the speed, accuracy, and memory requirements of the finger walker system.

The size of the lookup table affects both the accuracy and the memory requirements of the finger walker system. The current analog to digital card and EFPS have an effective acquisition range of 0 to 10 volts. The size of the table corresponds to the accuracy of the system according to how much of the analog to digital data is used. For example, a size 11

table will be accessed by integer voltage values, each corresponding to a different transmitter position (i.e. V=1.9762 accesses table location 1). On the other hand, a size 101 table will access different transmitter positions every 0.1 volts (i.e. V= 1.9762 accesses table location 19). Increasing the size of the lookup table increases the resolution of the finger walker system. The problem with increasing the table size is that each increase in size requires a proportional, linear increase in system memory:

$$\theta(IN)\ , \tag{17}$$

```
#define COL             11
#define ROW             11

// initializing the lookup table
struct coord {
        double x;
        double y;
};

// lookup[X][Y]
 struct coord lookup_1[ROW][COL]={
{{4,4}, {4,3.5}, {4,3}, {4,2.5}, {4,2}, {4,1.5}, {4,1.25}, {4,1.25}, {4,1}, {4,.5}, {4,0}},
{{3.5,4}, {3.5,3.5}, {3.5,3}, {3.5,2.5}, {3.5,2}, {3.5,1.5}, {3.5,1.25}, {3.5,1.25}, {3.5,1}, {3.5,.5}, {3.5,0}},
{{3,4}, {3,3.5}, {3,3}, {3,2.5}, {3,2}, {3,1.5}, {3,1.25}, {3,1.25}, {3,1}, {3,.5}, {3,0}},
{{2.5,4}, {2.5,3.5}, {2.5,3}, {2.5,2.5}, {2.5,2}, {2.5,1.5}, {2.5,1.25}, {2.5,1.25}, {2.5,1}, {2.5,.5}, {2.5,0}},
{{2,4}, {2,3.5}, {2,3}, {2,2.5}, {2,2}, {2,1.5}, {2,1.25}, {2,1.25}, {2,1}, {2,.5}, {2,0}},
{{1.5,4}, {1.5,3.5}, {1.5,3}, {1.5,2.5}, {1.5,2}, {1.5,1.5}, {1.5,1.25}, {1.5,1.25}, {1.5,1}, {1.5,.5}, {1.5,0}},
{{1.25,4}, {1.25,3.5}, {1.25,3}, {1.25,2.5}, {1.25,2}, {1.25,1.5}, {1.25,1.25}, {1.25,1.25}, {1.25,1}, {1.25,.5},
{1.25,0}},
{{1.25,4}, {1.25,3.5}, {1.25,3}, {1.25,2.5}, {1.25,2}, {1.25,1.5}, {1.25,1.25}, {1.25,1.25}, {1.25,1}, {1.25,.5},
{1.25,0}},
{{1,4}, {1,3.5}, {1,3}, {1,2.5}, {1,2}, {1,1.5}, {1,1.25}, {1,1.25}, {1,1}, {1,.5}, {1,0}},
{{0.5,4}, {0.5,3.5}, {0.5,3}, {0.5,2.5}, {0.5,2}, {0.5,1.5}, {0.5,1.25}, {0.5,1.25}, {0.5,1}, {0.5,.5}, {0.5,0}},
{{0,4}, {0,3.5}, {0,3}, {0,2.5}, {0,2}, {0,1.5}, {0,1.25}, {0,1.25}, {0,1}, {0,.5}, {0,0}}
};

{
        first = lookup_1[temp[0]][temp[1]];

        second = lookup_2[temp[2]][temp[3];
}
```

Figure 19: The C++ code to initialize and access the two dimension eleven by eleven lookup table shown in figure 18.

where the system memory is a function of the size of the variable used to store the position vector (8 or 16 bytes depending on whether a single coordinate or a coordinate pair is stored), I, and the size of the lookup table, N.

The number of dimensions used in the lookup table affects both the speed and the accuracy of the system. The current finger walking interface calls for two transmitters and four receivers resulting in two voltage values for each transmitter. Thus, there are three possibilities for the table dimension used: one, two, and four dimensions. The number of processor cycles required by the lookup table is negatively affected by increases in the number of tables required:

$$\theta(X)\,, \tag{18}$$

where the number of processor cycles required is a function of the number of tables, X. For example, a four dimensional table requires only one memory access, while four, one dimensional tables require four memory accesses. On the other hand, the memory requirements of the lookup tables are positively affected by increases in the number of dimensions:

$$\theta\left[(5-D)N^{D}\right], \tag{19}$$

where the memory required is a function of the number of dimensions, D, and the size of the lookup table, N. For example, a four dimensional 100 entry table requires 800,000,000 bytes, while four, one dimensional 100 point tables require 3200 bytes.

These factors affecting system performance must be weighed against each other in order to determine the best possible combination. Through experimentation and analysis of

the data from Experiment 1 (section 5), it was determined that the best lookup table configuration consists of four, one dimensional size 101 tables. This configuration is fairly fast, requires four memory lookups, has an accuracy of 0.1 volts, and uses only 3200 bytes of system memory.

Once the size and number of dimensions of the lookup table are determined, the position vector must be preprocessed. The test program Tbed1.c is run with the final finger walker configuration to determine the voltage vs. position of the system. This data is then graphed and a table of the positions (x, y) is created which is referenced by the voltages. The accuracy of the lookup table is increased and the memory usage is decreased with this interpolation of the position and electric potential experimental data.

6.3 Performance

Both methods, processing the position vector mathematically in real time or preprocessing it in a lookup table, have several advantages and disadvantages. A comparison between these two methods is made by examining the memory requirements, resolution, and processing speed of each method.

6.4.1 Memory Requirements

Since system memory in computers is easily expandable to meet almost any need, the memory requirements of the finger walker software are not very important. However, it is still desirable to keep the requirements to a minimum. Processing the data in real time requires only the memory to hold the code instructions and a few temporary variables. On

the other hand, the lookup table requires approximately 3200 bytes of memory for storing the necessary data.

6.4.2 Resolution

As discussed in section 5, a crucial factor in the design of the finger walker is system resolution. The finger walker needs to be accurate enough to detect small changes in the user's finger positions in order to determine their movement. The real time processing method gives extremely good accuracy as compared to the lookup table. Real time processing can use the entire range of voltages received without changing the speed of the system (0.005V). However, with a large enough lookup table, the difference in resolution between real time processing and the lookup table may not be noticeable to a user. A lookup table with a resolution comparable to real time processing would require 2001 entry points per receiver.

6.4.3 Speed

The most critical factor affecting the performance of the finger walker interface is speed. The interface must update the data fast enough for movement through the VE to run smoothly. In terms of speed, there is no question that the lookup table far outperforms the real time method. The lookup table takes only a few clock cycles to access the preprocessed data in memory, while processing the data in real time requires hundreds of floating point operations which require hundreds of clock cycles. Therefore, for the purposes of this project, the lookup table is used for determining the movement of the user's fingers over the pad.

7 Determining The Velocity Vector

With a position vector determined for each of the user's two fingers, all that remains is to determine the velocity vector of the movement. The first stage of this process is to calculate the velocity of each of the fingers (section 7.1). Next, these two vectors need to be translated into a movement vector for the user (section 7.2), which dictates the movement of the user through the virtual environment. Using the results of section 7.2, the system can determine the overall magnitude (section 7.3) and direction (section 7.4) of the user through the virtual environment.

7.1 Finger Velocities

The first step in processing the position vectors collected from the electric field sensing is to determine the velocity of each finger. Figure 20 shows the current position of the user's fingers, $P_1\,(x_{P1},\, y_{P1})$ and $P_2\,(x_{P2},\, y_{P2})$, along with the past position of the fingers, $N_1\,(x_{N1},\, y_{N1})$ and $N_2\,(x_{N2},\, y_{N2})$, during a normal walking motion. An estimation of the velocity of each finger, V_1 and V_2, is determined by comparing the past positions with the current positions:

$$\begin{aligned} V_1 &= (x_{P1} - x_{N1})\hat{i} + (y_{P1} - y_{N1})\hat{j} \\ V_2 &= (x_{P2} - x_{N2})\hat{i} + (y_{P2} - y_{N2})\hat{j} \end{aligned}, \qquad (20)$$

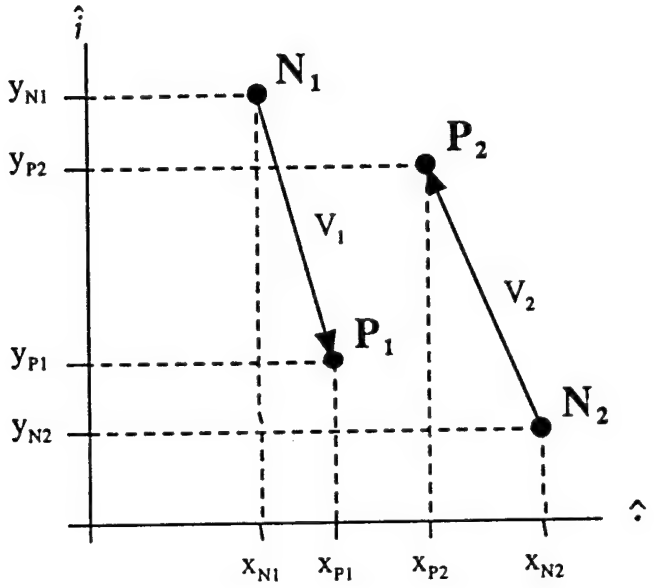


Figure 20: Determining the finger velocities.

where $\hat{i}$ is the unit vector in the x direction and $\hat{j}$ is the unit vector in the y direction.

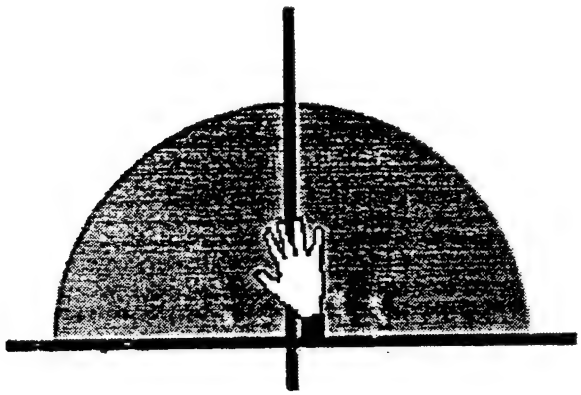


Figure 21: Arc of movement dictated by the human wrist. The user can move forward only in the shaded region.

7.2 Determining A Movement Vector

The next step in the process is to determine an overall velocity vector for the user from the velocity vectors of the individual fingers. There are several different movements which need to be examined: forward movement, reverse movement, turning left, and turning right. Because of the limitations of the human wrist, the user can only make a forward motion within in a 180 degree arc (see figure 21).

During normal forward motion, one finger will be moving forward with the direction of movement, and one finger will be moving backwards. Thus, the normal walking motion will consist of one positive and one negative velocity. The movement forward corresponds to the user's finger moving forward through the air to take a step, while the movement backward corresponds to the user's finger sliding across the pad. Both fingers are moving at approximately the same rate and direction. In order to simplify the algorithm used to determine the user's movement through the virtual environment, the velocity of the user's finger sliding on the pad will be used as the overall movement velocity. Therefore, only the velocity in the reverse direction (V_2 in figure 22a and V_1 in figure 22b) is used to determine the forward movement of the user through the virtual environment.

Because the sensor cannot differentiate between finger walking forward (accomplished by one finger moving forward and one back) and finger walking backward (also with one finger moving forward and one backwards), there must be a unique motion which corresponds to the reverse movement of the user. In the case of the finger walker, this unique motion is both of the user's fingers moving in the reverse direction. Therefore, both finger velocities, V_1 and V_2 in figure 22c, are used to determine the reverse movement of the user through the virtual environment.

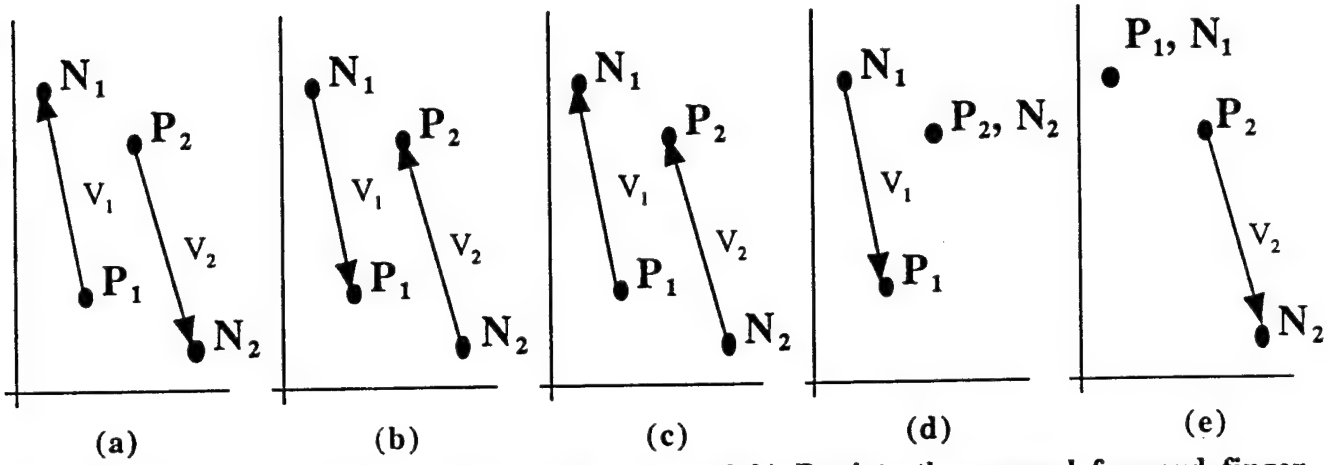


Figure 22: Examples of finger velocities. (a and b) Depicts the normal forward finger walking motion. (c) Depicts both fingers moving in the same direction. When these vectors are positive the forward which does not effect the user's movement through the virtual environment. When they are negative the user moves backwards through the environment. (d) Depicts a left turn. (e) Depicts a right turn.

Since the user can only comfortably make a walking motion with his fingers in a 180 degree arc, the tracking system requires a method to turn past the limits of this arc. In the case of the finger walker, the turning movement will correspond to the movement of only one finger while the other finger remains stationary. A left turn will be represented by the movement of the left finger, V_1 in a walking motion (see figure 22d), while a right turn will be represented by the movement of the right finger, V_2 in a walking motion (see figure 22e).

7.3 Magnitude

Virtual environments require the magnitude of the user's velocity from the user interface in order to update the current position of the user. Once the velocities of the individual fingers have been determined, it is simple to calculate this magnitude. Using the

individual finger velocities from equation 20 and the simplifications from section 7.2, the magnitude of the movement through the virtual environment is

$$|V_{total}| \begin{cases} = \sqrt{\left(V_1 \hat{i}\right)^2 + \left(V_1 \hat{j}\right)^2} & \text{(Forward, if } V_1 \text{ is Negative)} \\ = \sqrt{\left(V_2 \hat{i}\right)^2 + \left(V_2 \hat{j}\right)^2} & \text{(Forward, if } V_2 \text{ is Negative)} \\ = \dfrac{\sqrt{\left(V_1 \hat{i}\right)^2 + \left(V_1 \hat{j}\right)^2} + \sqrt{\left(V_2 \hat{i}\right)^2 + \left(V_2 \hat{j}\right)^2}}{2} & \text{(Reverse)} \\ = 0 & \text{(Turning Left)} \\ = 0 & \text{(Turning Right)} \end{cases} \quad . (21)$$

When one finger is moving forward and the other backwards, the magnitude of the user is the resulting vector formed by the $\hat{i}$ and $\hat{j}$ components (the hypotenuse of the right angle triangle formed from the two components). When both fingers are moving backwards, the magnitude is the average of the two vectors formed by the movement of each finger. There is no magnitude when the user is turning in place.

7.4 Direction

Virtual environments also require the direction of the user's velocity from the user interface in order to update the virtual environment with the current position of the user. Once the velocities of the individual fingers have been determined, it is simple to calculate the direction the user is traveling through the virtual environment. Using the individual

finger velocities from equation 20 and the simplifications from section 7.2, the magnitude of the movement through the virtual environment is

$$Direction = \begin{cases} = \tan\left(\dfrac{V_1 \hat{j}}{V_1 \hat{i}}\right) & \text{(Forward, if } V_1 \text{ negative)} \\ = \tan\left(\dfrac{V_2 \hat{j}}{V_2 \hat{i}}\right) & \text{(Forward, if } V_2 \text{ negative)} \\ = \tan\left(\dfrac{\left(V_1 \hat{j} + V_2 \hat{j}\right)}{\left(V_1 \hat{i} + V_2 \hat{i}\right)}\right) & \text{(Reverse)} \\ = \sqrt{\left(V_1 \hat{i}\right)^2 + \left(V_1 \hat{j}\right)^2} * (TurningMultiplier)(2\pi) & \text{(Turning Left)} \\ = \sqrt{\left(V_2 \hat{i}\right)^2 + \left(V_2 \hat{j}\right)^2} * (TurningMultiplier)(2\pi) & \text{(Turning Right)} \end{cases} \quad . (22)$$

When one finger is moving forward and the other backwards, the user's direction is the angle formed by the two components of movement. If both fingers are moving backwards, the direction is the average of the angles formed by each of the finger movements. Finally, when only on finger is moving, the user turns at a rate depending on the magnitude of the finger moving times the Turning Multiplier and 2π.

8 System Architecture

The next step in the design of an operational Finger Walker user interface is the construction of an operational prototype. As mentioned in section 2, the ideal Finger Walker will incorporate the following design criteria into its construction:

-Expandable to Slippery Floor

-Low cost

-Require minimal equipment on user

-Robust

-Accurate

-Reliable/Repeatable

-Ease of use

The prototype finger walker will integrate as many of these design criteria as possible. Section 8.1 gives an overview of the integrated system, including a discussion of the steps in its operation. The design of the prototype can be separated into two separate system, the hardware system (section 8.2) and the software system (section 8.3). Virtual environments will access the integrated system through a set of C/C++ library functions (section 8.4).

8.1 Integrated System Overview

As previously mentioned, the finger walking device will be an inexpensive, easy to use, interface for virtual environments. The operator of the finger walker will use a natural

walking-like motion with his fore and middle fingers with minimal equipment attached to the user's body. The input to the finger walker user interface will be a tracking of the change in the electric field created by the user's fingers. The Finger Walker will output a velocity vector, which will consist of a magnitude and a direction, to the virtual environment.

The operation of the finger walker user interface is easy and straight forward. First, the user will sit down at the computer, workstation, or other location setup for viewing the virtual environment. The user then attaches the transmitter electrodes to his fingers for tracking. Next, the user will place a HMD on his head or position himself before a standard computer monitor to view the virtual environment. Finally, the user will place his fingers on the finger walker pad and begin moving his fingers in a walking-like motion. The finger walker and the virtual environment software perform the calculations which update the position of the user in the virtual environment. The finger walker user interface consists of five distinct stages of operation: signal detection, data acquisition, translation, special operation instructions, and velocity computation.

Signal detection and data acquisition are both formed in the finger walker hardware. The hardware consists of the finger walker pad, Electric Field Proximity Sensor boards, power supply, and the analog to digital converter. As the user moves his fingers over the pad, the position of the transmitter changes. The transmitter emits an electric field which is detected by the receivers. The four receivers detect the distance to the transmitter by the strength of the signal at the point of detection. This strength is represented as an electric potential, i.e. analog voltage level. This voltage will be sent to the computer running the finger walker software.

Next, the voltages detected by the receivers are sent to a computer for processing. The analog signals from the receivers must be converted to digital signals which can be interpreted and manipulated by a computer. Signal conversion can be accomplished through the use of an analog to digital converter card which connects to the system bus of most PCs. This process will allow the processor to directly access the data acquired by the receivers. The analog electric potential from the Electric Field Proximity Sensor board is now available for manipulation by software into a position vector.

The translation, special instructions, and velocity computation stages are all performed in the finger walker software system. Once it is available as a digital value from the analog to digital converter card, the electric potential voltage can be converted to a distance vector in meters. This translation is accomplished in a memory lookup table which accesses a specified memory location containing the proper coordinates of the position of the finger on the pad. The memory address accessed depends on the magnitude of the voltage received from the analog to digital converter card.

Once the coordinates of each individual finger have been calculated, the software then determines the type of movement and any special instructions contained in the movement. The new coordinate pairs from the lookup table can be compared to the previous set of coordinates to determine the change in position of each finger. By analyzing the velocities of each finger, the specific type of movement can be determined. For example, a normal forward motion would consist of one finger moving in the positive y direction and the other finger moving in the negative y direction, see figure 23. Currently the system recognizes normal forward movement (both fingers moving in opposite directions), reverse movement (both fingers moving backwards at the same time), turning left or right (one finger stationary and the other moving), or no movement at all (both fingers moving forward or no change in position).

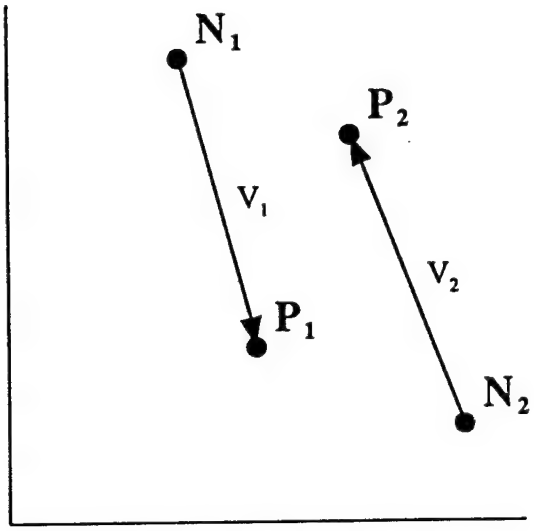


Figure 23: Normal walking motion.

With the type of movement determined, the software computes the velocity vector for the virtual environment. The virtual environment software requires a velocity vector in order to change the position of the user in the virtual environment. The type of movement determined by the software dictates which finger velocity or velocities to use as the overall movement velocity. This movement velocity can then be changed into a magnitude and direction and sent to the virtual environment software.

8.2 Hardware Systems

The hardware systems detect the electric field and send the data to a computer for processing. There are three main hardware subsystems: the pad (section 8.2.1), the proximity sensors (section 8.2.2), and the analog to digital card (section 8.2.3).

8.2.1 Pad (Transmitters and Receivers)

The pad subsystem is the critical component of the finger walker human-computer interface. The subsystem consists of the walking surface, the transmitters, and the receivers. The same test platform as used in experiment 1 (section 5) is utilized in the final system prototype (appendix B). As described, two thimble type transmitters are used to track the position of the separate fingers. Finally, as discussed in section 5.6, the optimal choice for the receiver array are four, 8" long, 18 gauge wires mounted 1-1/2" from the edge of the pad.

8.2.2 Electric Field Proximity Sensor System

The Electric Field Proximity Sensor subsystem is the same as in Experiment 1 (section 5). Two pairs of EFPS boards are used for each transmitter. The wiring diagrams can be seen in Appendix C.

8.2.3 Analog To Digital Card

As with experiment 1 (section 5), the Keithley DAS-1602 analog to digital card is used for converting the analog electric potential from the EFPS to a digital value for computing the position of the user's fingers. The configuration files and the wiring diagram for the connection between the analog to digital card and the EFPS can be found in Appendix C.

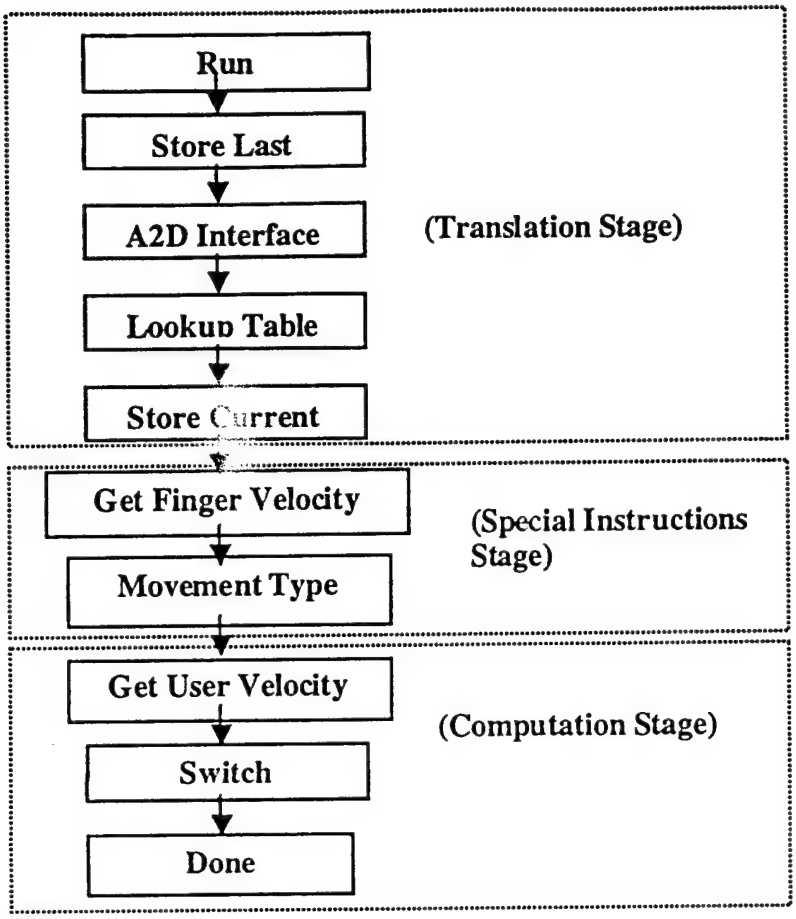


Figure 24: Flow chart of operation for the finger walker software.

8.3 Software Systems

The remaining three stages of processing are performed by means the finger walker software package. The software manipulates the electric potential received from the analog to digital card to compute a velocity vector. These program components include the translation of the data (section 8.3.1), the determination of special instructions (section 8.3.2), and the computation of a velocity vector (section 8.3.3). The overall program flow of the software system can be seen in figure 24. The compete software package code can be seen in Appendix G.

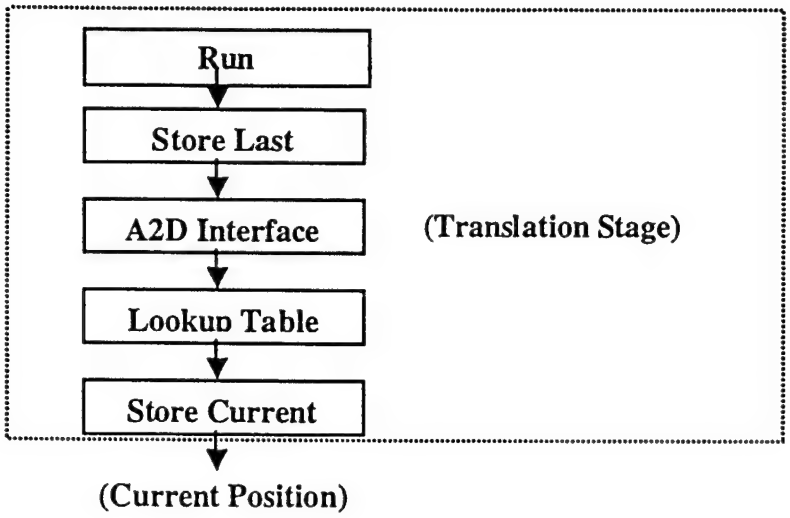


Figure 25: Flow chart of the software translation stage.

8.3.1 Translation

The translation of the electric potentials from an analog voltage to a set of coordinates representing the current position of the fingers is the first of three software components. This section of the software applies the principles from sections 4, 6, and 7. The translation stage can be separated into five main functions: the storage of previous points, the analog to digital control, the lookup table, the storage of the current points, and error detection. A flow chart of the translation stage is shown in figure 25.

Before the analog to digital card can be accessed and an electric potential gathered, the previous set of coordinates must be stored. The previous coordinates are stored in two coordinate structures, *old_first* and *old_second*. The function *store_last()*, as shown in Appendix , performs this task. The variables are initially set to (2, 2) and (4, 2) respectively.

The analog to digital card is controlled through the software functions provided in the ASO-1600 Dasshl32.lib library. The program code for accessing the DAS-1600 is shown in Appendix G. The final voltages are stored in the global array *data[4]* (*data[0]* holds the voltage received at receiver 1, *data[1]* corresponds to receiver 2, etc.).

These four voltages are then used to access the memory lookup table. In the finger walker prototype, four 101-element arrays of coordinates are used for the lookup tables, one for each receiver. Table 1 is accessed by sending it the value detected at receiver 1, table 2 is assessed by the value at receiver 2, etc. See Appendix G for the program code showing the access of the lookup tables. Each memory address contains the position of the one finger in either the x or the y direction depending on the receiver. For example, table 1 and table two contains the x and y values of finger one. Because a memory address is an integer, the decimal point must be stripped off of the voltage value and all the data to its right discarded. Thus, each voltage must be multiplied by 10 in order to maintain the correct number of significant digits and use the full precision of the array.

The storage of the finger positions is performed after each table is accessed. See Appendix G for the code. The coordinates of each finger are stored in the global array *coords[4]* for easier processing by the remaining stages of the software.

Each of the main sections of code (Translation, Special Instructions, and Computation) contains error detection code. In the event of an error, the software will determine whether it needs to restart the entire process to get another set of voltages or begin a single function, or set of functions, over. The analog to digital function contains code to check the range of values received.

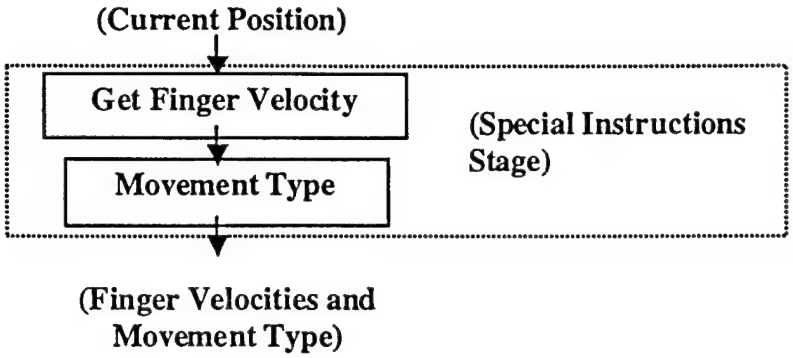


Figure 26: Flow chart of the special instructions software stage.

8.3.2 Special Instructions

The second of the three software components, special instructions, analyzes the position vectors from the translation stage in order to determine the type of movement of the user. This section of the software applies some of the principles from section 3. The special instructions stage consists of two key functions: determining the x and y component of each fingers velocity, and determining the movement type. A flow chart of the special instructions stage is shown in figure 26.

First, the position vectors from the translation stage must be analyzed in order to determine the x and y components of the velocity for each individual finger. The global position structures *first*, *old_first*, *second*, and *old_second* are used to determine the velocity of finger one and finger two (see Appendix G for the code). The functions *Get_Delta_X1()*, *Get_Delta_X2()*, *Get_Delta_Y1()* , and *Get_Delta_Y2()* determine the change in position of the fingers. These velocity components are stored in the global variables *Delta_X1*, *Delta_X2*, *Delta_Y1*, and *Delta_Y2* for later use by the this stage and the computation stage.

Finally, the finger velocities are analyzed through a set of nested if statements to determine the type of movement or special instruction. Currently, there are five different types of instructions: normal forward movement (0), nothing (1), reverse movement (2), a right-turn (3), and a left-turn (4). Because of the movement restriction imposed by the user's wrist, only the y components need to be analyzed, simplifying the analysis. The nested if statement, shown in Appendix G, processes the four components of the finger velocities by analyzing the positive or negative motion of the y component of each finger's velocity (*Delta_Y1* and *Delta_Y2*). For example, if *Delta_Y1* is positive and *Delta_Y2* is negative the user is moving his fingers in a normal walking motion with one finger moving forward and one moving back. Table 3 shows a complete list of finger velocities and the corresponding movement types. This program format allows for new instructions to be easily added upon demand. The movement type is stored in the global variable *type* for use by the computation stage.

Finger One	Finger Two	Movement Type
none	none	Nothing (1)
none	positive	Nothing (1)
none	negative	Right-Turn (3)
positive	none	Nothing (1)
negative	none	Left-Turn (4)
positive	negative	Normal (0) Finger 2
negative	positive	Normal (0) Finger 1
positive	positive	Nothing (1)
negative	negative	Reverse (2)

Table 3: Listing of all possible finger movement combinations.

8.3.3 Computation

The final software component, computation, analyzes the finger velocities and movement type from the special instructions software component in order to determine the velocity of the user through the virtual environment. This section of the software applies the principles from section 7. The computation stage consists of three main functions: the determination of the x component of the velocity, the determination of the y component of the velocity, and the switch statement, which outputs the magnitude and direction for the virtual environment. A flow chart of the computation stage is shown in figure 27.

First, the computation stage calls two functions, *Get_Velocity_Delta_X* and *Get_Velocity_Delta_Y*, which use the movement type from the special instruction software component to determine the magnitude and direction of the user's movement. The movement type specifies whether to use the velocity of finger one, finger two, or both in order to determine the x and y components of the user's movement.

Next, the switch statement sets the magnitude and direction of the user's movement through the virtual environment. These variables completely depend on the movement over

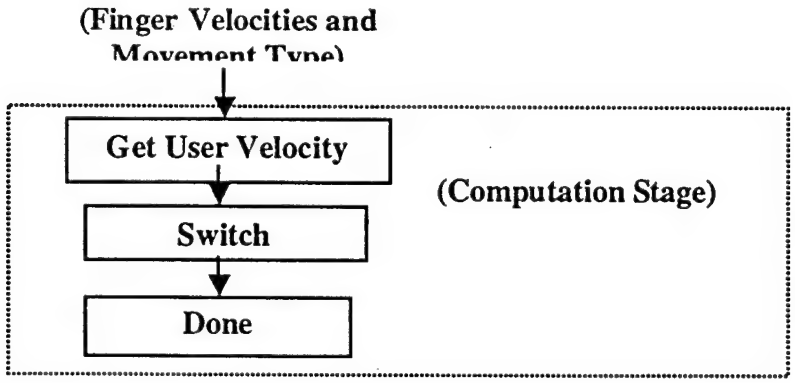


Figure 27: Flow chart of the software computation stage.

the pad as discussed in section 7. Each time the users makes either a right or left turn, the magnitude of the turn is added to a turning offset (*turning_offset*) which is applied to all future movements.

Finally, the magnitude and direction set in the switch statement can be modified to adjust the rate of movement and turning within the virtual environment. The two parameters, *turning_multiplier* and *forward_multiplier*, are scaling factors for modifying the user's change in direction and forward magnitude. The correct settings for these parameters depend on the specific virtual environment.

8.4 Library Functions

The software functions from section 8.3 are incorporated into a C++ library, *finger.hpp* and *finger.cpp* (see Appendix G for a complete listing of the library). These library functions and files are easily implemented. To use the functions, the header file is included in the virtual environment software programs:

```
#include "finger.hpp"
```

This library allows programs to retrieve data in two ways: through the return value of the functions or through the global variables (*Type*, *Magnitude*, *Direction*, etc). Finally, this library includes the *F_Run()* and *F_Run_Debug* functions for accessing the finger walker software.

The *F_Run* library function is used to access the finger walker hardware and set all the global variables depending on the movement of the user's fingers. The function prototype for this function is

```
int F_Run(int CallNumber);
```

It takes the call number as an input parameter. The first time the Finger Walker interface is accessed, the programmer should send a 0. Otherwise the programmer can send any other number. When a 0 is passed, the program calls the hardware twice in order to determine the magnitude and direction. This first call to the hardware sets the old coordinate variables and the second call sets the present set of coordinates. With this done, the magnitude and direction can be determined. This function outputs an integer variable representing the type of movement by the user's fingers: normal (1), reverse (2), left-turn (3), right-turn (4), and nothing (0). When called this function uses the functions described in section 8.3. This function also updates all the global variables associated with the finger walker.

The *F_Run_Debug* library function is used to access the finger walker software and set all the global variables depending on the parameters set. The function prototype for this function is

```
int F_Run_Debug(void);
```

This function bypasses the hardware by calling a lookup table with specific voltages depending on the *Depug_Number*, a global variable. A debug number of 0 causes a normal walking motion, 1 causes nothing, 2 causes a reverse walking motion, 3 causes a left turn, 4 causes a right turn, and 5 causes an angled forward walking motion. Next, the function sets the present set of coordinates, then determines the magnitude and direction.

This function outputs an integer variable representing the type of movement by the user's fingers: normal(1), reverse(2), left-turn(3), right-turn(4), and nothing(0). When called this function uses the functions described in section 8.3. This function also updates all the global variables associated with the finger walker.

9 Experiment #2—Demonstration Program

In order to demonstrate the library functions and the finger walker user interface detailed in section 8, a Windows95 application was created, ADTracker. This application also allowed the entire system to be debugged. The finger walker library functions are controlled by the Demo Application through several control buttons (section 9.1). This application allows the coordinates, magnitude, direction, and type of movement to be graphically displayed (section 9.2) using a coordinate grid (section 9.3), a compass (section 9.4), and a tracking window (section 9.5). A screen shot of the demo application can be seen in figure 28. The code for this application is given in Appendix H.

9.1 Controls

The finger walker library functions are controlled by the Demo Application through several control buttons and a parameters window. The control buttons include a *Run*, *Stop*, *Debug*, and *Quit* control the flow of the ADTracker program. The parameters window, which is accessed through the control button *Parameters*, allows the user to adjust several variables of the finger walker software.

The user of the demo program has the ability to control the finger walker through several control buttons. When *Run* is pressed, the program begins a loop which continually calls the *F_Run()* function and updates the windows until the user presses *Stop*. This demonstration program also gives the user the ability to run a diagnostic program which sends one of five different preset walking patterns to the ADTracker program, testing the functionality of the ADTracker program and the finger walker software from sections 8 and 9. The *Quit* button simply exits the program.

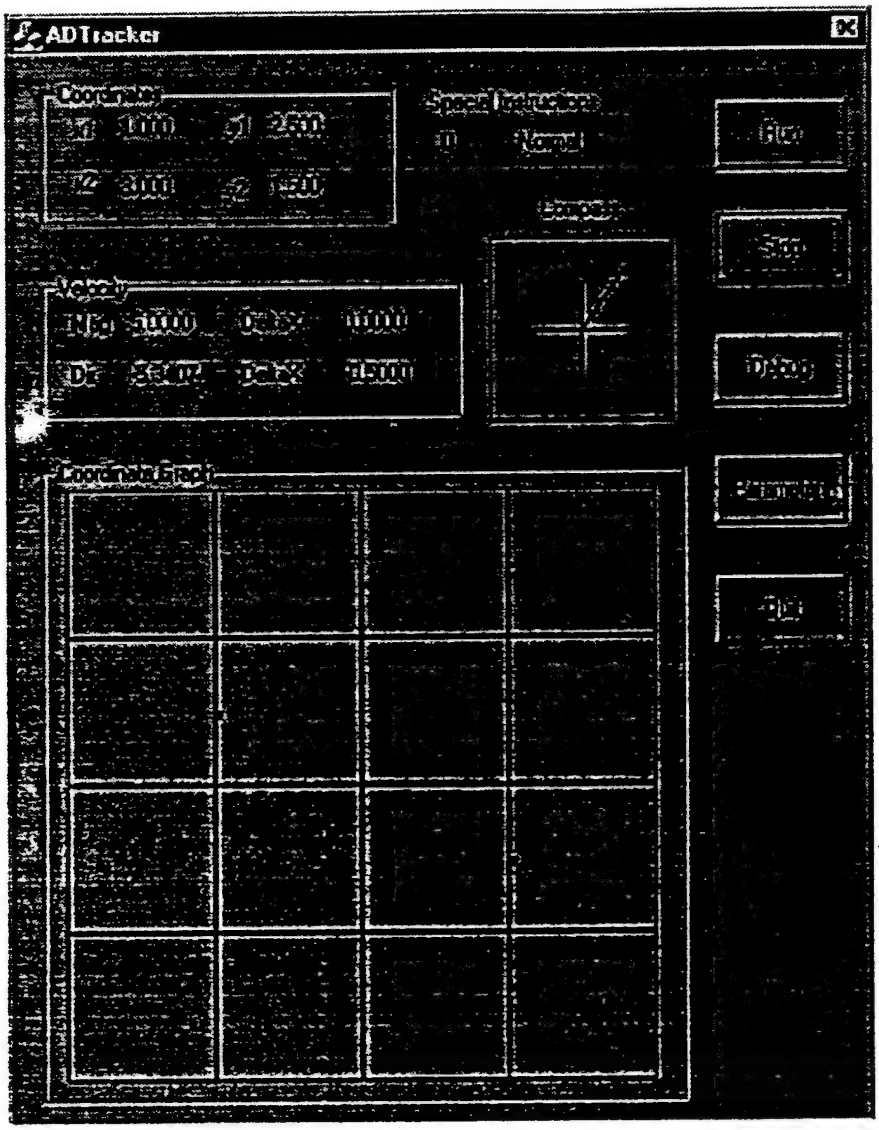


Figure 28: Main program screen for the ADTracker application.

Between operations, the user can adjust several variables, *Turning_Multiplier*, *Forward_Multiplier*, and *DebugNumber* in order to calibrate the system. The parameters are adjusted through the pop-up window shown in figure 29. The *Forward_Multiplier* and *Turning_Multiplier* variables may be set to any floating point number. The *DebugNumber* is set to any integer between 0 and 5: normal walking (1), reverse walking (2), nothing (0), right-turn (3), left-turn (4), and angled forward walking (5).

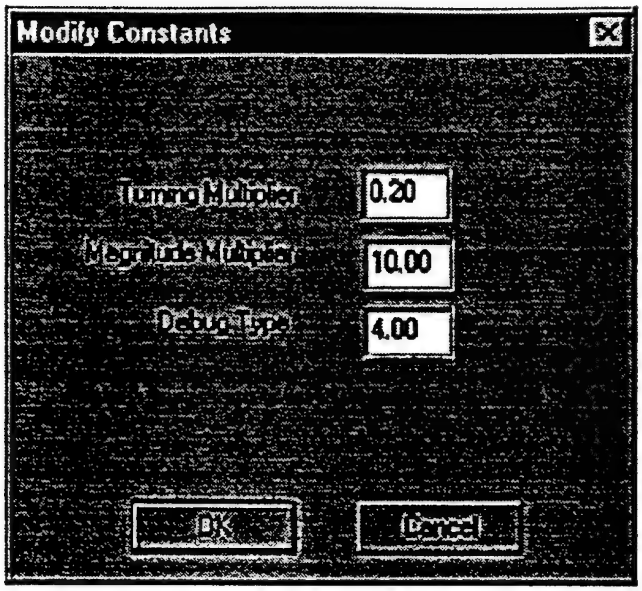


Figure 29: Parameters popup window.

9.2 Display

The global variables contained in the finger walker software are displayed within the ADTracker application window during operation (see the screen shot in figure 28). The x and y coordinates of the two fingers are shown in the upper left corner of the application window (*first.x*, *first.y*, *second.x*, and *second.y*). The velocity of the user through the virtual environment (Magnitude, Direction, *Delta_Velocity_X*, and *Delta_Velocity_Y*) is displayed directly below the displayed coordinates. Finally, the type of motion of the fingers is displayed in the top center. With these variables displayed, the position and movement of the user can be tracked at all times.

9.3 Position Map

The coordinates of the user's fingers is graphically displayed in a position map. By displaying the coordinates (*first.x*, *first.y*, *second.x*, and *second.y*) graphically, the finger walker software can be debugged. The map consists of a 4" by4" grid located below the global variables (see the screen shot in figure 28) with two red dots representing the position of each of the user's fingers. This grid allows the user to place his fingers on the grid and check how well the coordinates generated by the finger walker interface match with the actual position on the grid.

9.4 Direction Compass

Just as the position map displays the finger coordinates graphically, the compass displays the walking direction of the user graphically. The compass can be seen in the screen shot shown in figure ??. The arrow points in the direction the user is moving or facing, when standing still. Because the finger walker allows the user to make left and right turns, the user's fingers may be moving in a different direction then the user is actually moving through the virtual environment. This offset (*direction_offset*) is displayed in the compass.

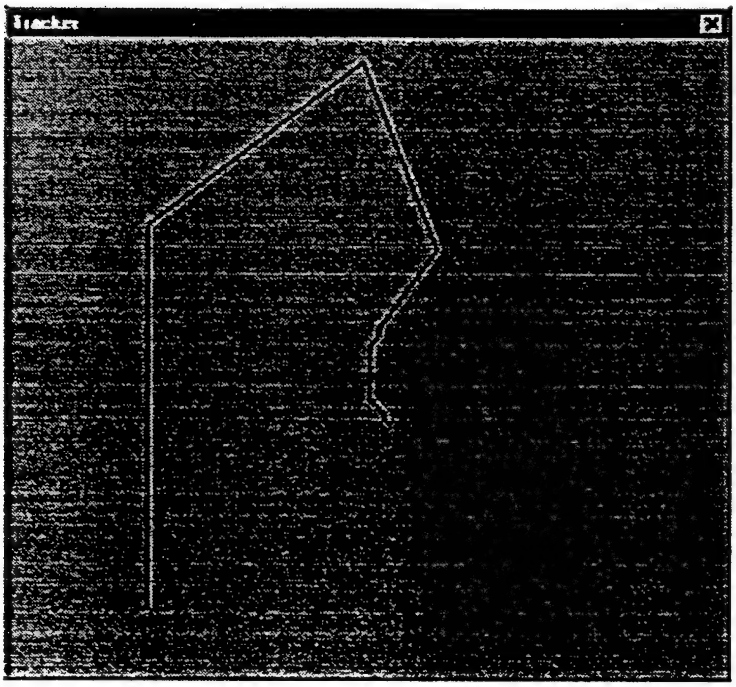


Figure 30: Tracker window.

9.5 Tracker

Finally, a separate window tracks the movement of the user through the virtual environment. This window draws a line along the path of the user as determined by the magnitude and direction global variables, see figure 30. When the user reaches the limits of the window, the ADTracker program scales the movement down, allowing the path room to expand.

10 Results/Conclusion/Further Research

The initial tests of the finger walking interface with the ADTracker have provided substantial evidence to support the Electric Field Proximity Sensor as an efficient method by which to track the movement of the user. The finger walker is fairly accurate when tracking the position of the fingers across the pad. In addition, the tracking window proves that the finger walker is an effective means by which to move through a virtual environment. However, several problems have arisen during these early stages of testing and more tests need to be run before the finger walking interface can be used as an interface for virtual environments or even expanded to track the motion of a user's legs.

The main problem encountered during experimentation with the finger walker is the lack of a historicis, a memory of past system events. Currently, the system only keeps track of the present and previous set of coordinates. In addition, the hardware (analog to digital converter and EFPS) is very sensitive to slight changes in the position of the finger and noise in the system. These factors cause the system to jump back and forth between different rates and even types of movement. By adding more dynamics to the system, both a system of filters and a more robust velocity algorithm may be designed to fix this problem. The high frequency fluctuations caused by noise and unrealized movement can be eliminated through a series of low pass filters setup both before and after the lookup table. The rapid changes in movement can be eliminated through a more robust velocity algorithm which averages the user's velocity over a longer period of time. Both the filters and the new velocity algorithm require past coordinates pairs to be stored in memory for longer period of time. Thus, instead of only using the current and previous coordinates to determine the velocity, the system could use the current and last 10 sets of coordinate pairs. The same holds true for the system of filters.

Once these improvements are made to the finger walker, the system should be tested further to determine its effectiveness as an interface for virtual environments. In addition, the effects of the walking motion and effort expended on the user's ability to estimate distances and feeling of presence in the virtual environment should be tested.

Next, experimentation with a single transmitter should be preformed. Many of the experiments from section 5 will need to be rerun in order to determine the most effective means of detecting the position of both fingers using only one transmitter. This system may require the addition of several more receivers and EFPS boards in order to differentiate between the signals being sent from the two fingers.

Finally, once the finger walker system is finely tuned, the full-scale walking interface should be created. The expansion to a full-scale walking prototype requires a new receiver array. Again, the experiments from section 5 will need to be rerun. It will likely turn out that the system will work simple by expanding the receivers proportionally with the increase in the size of the walking surface. One benefit of the walking interface is the problem of not being able to turn 360 degrees is eliminated. In addition, the special instruction for finger walking in reverse is no longer required. However, the walker needs some method of determining the direction the user is facing, which is not required in the finger walker.

Acknowledgments

It is impossible to express my appreciation for all the help and guidance that everyone has given me. I am just ecstatic that this thesis project has finally come together and is complete.

I must first thank my thesis advisor Nat Durlach and Tom Wiegand for sticking with me and supporting me on this project.

Then there is my friend Evan Wies. He introduced me to Tom and helped me receive the project. He talked me though the concepts and ideas associated with virtual reality and haptics. He also spent endless hours proofing my final thesis helping me to finish on time (or at least close enough to graduate).

I really appreciate all the help Sam Pearlman gave me with teaching Microsoft Visual C++ to me. I could not have finished my thesis without your help with coding. Roads is on me for a while.

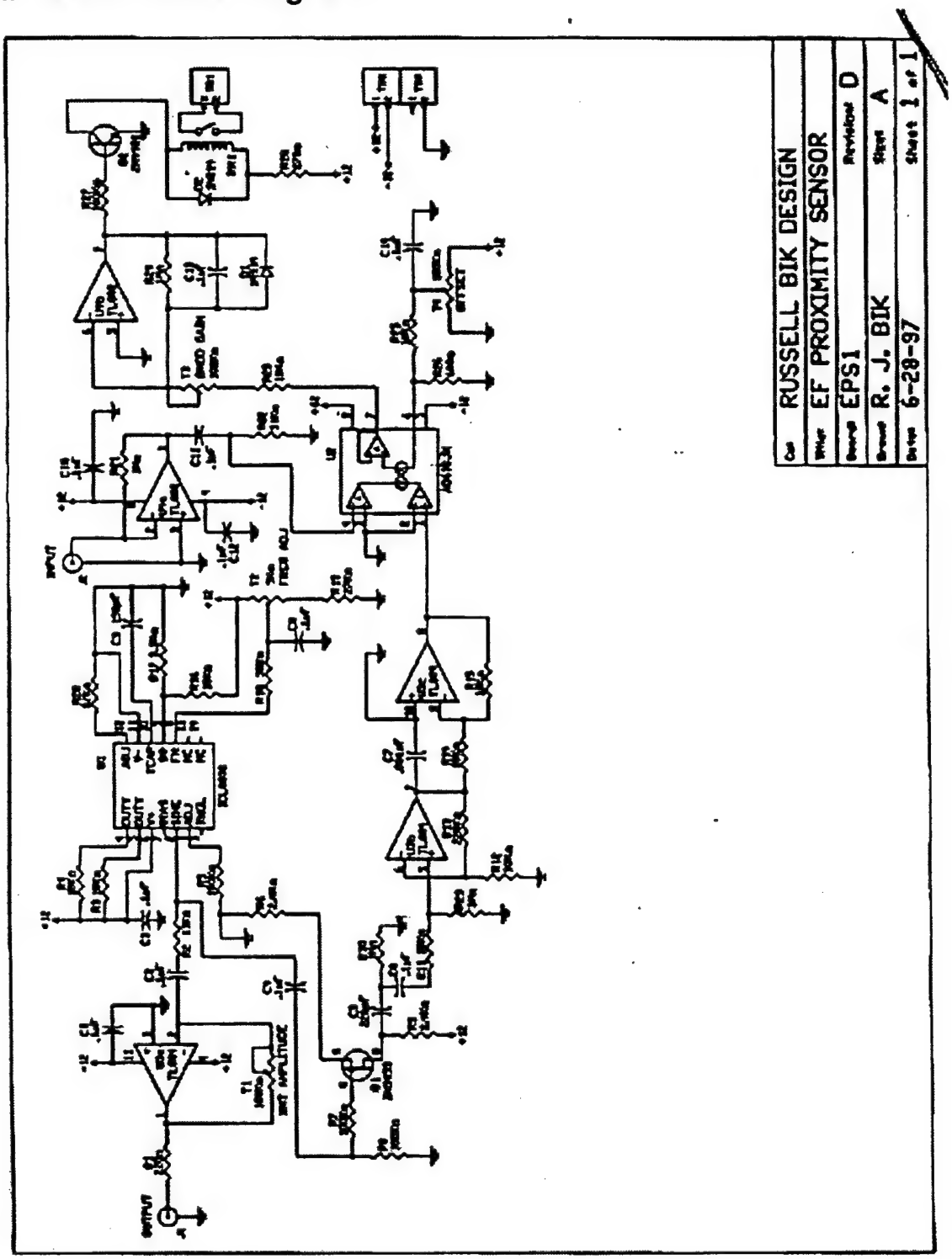
Abby Willets, Bryan Kincy, and Wayne Johnnie helped immensely by allowing me to work on their computers into the late hours of the night. Thanks Abby for all the support you have given me all the support and help I could possible need.

I must also thank my parents for their support and dedication to seeing me finish this project.

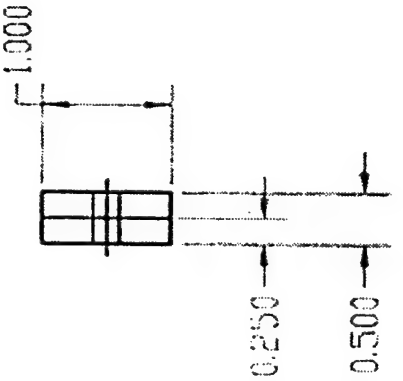
Finally, I'd like to thank the MIT Women's Polo Team for making my last term enjoyable and survivable. Congrats on such a good season.

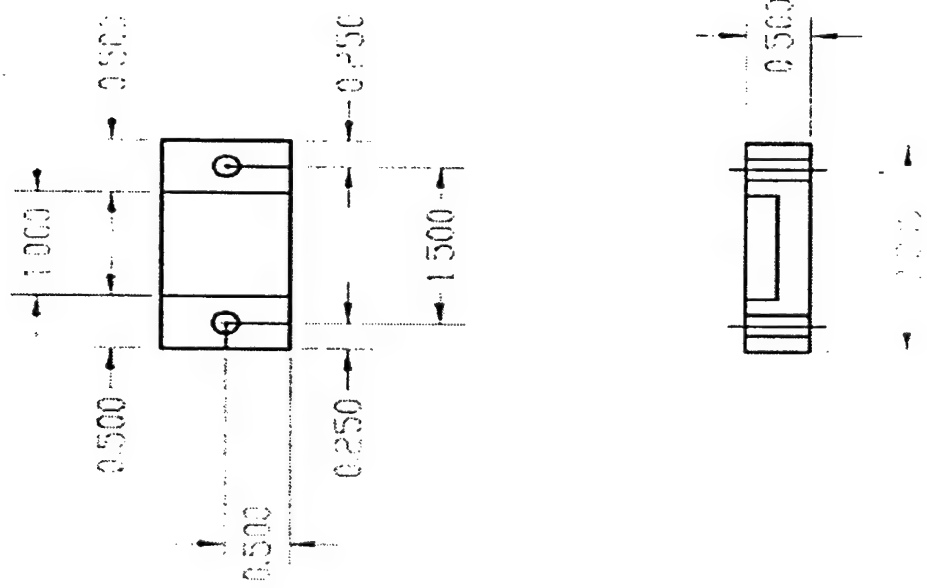
Appendices

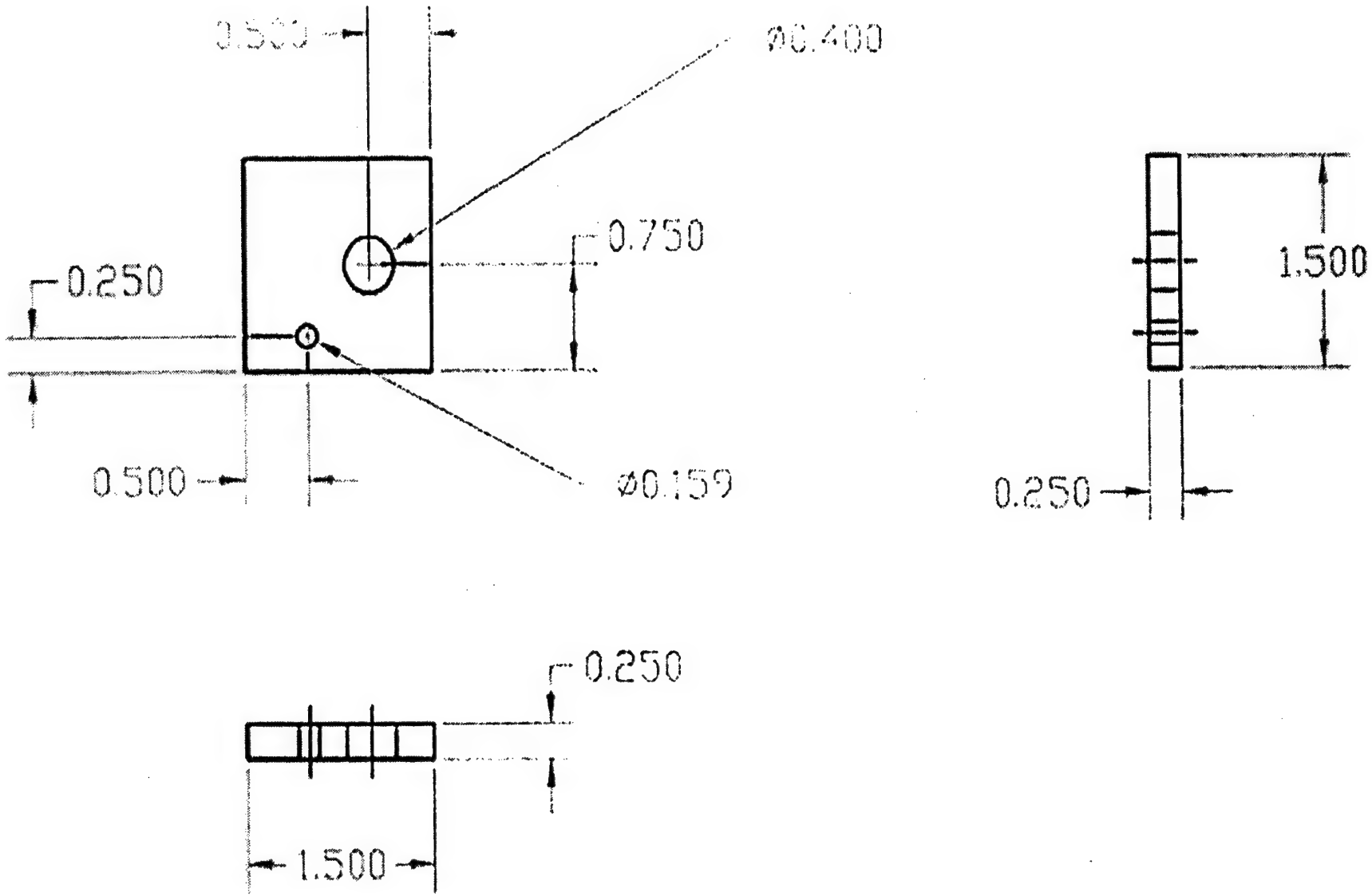
A. EFPS circuit diagram

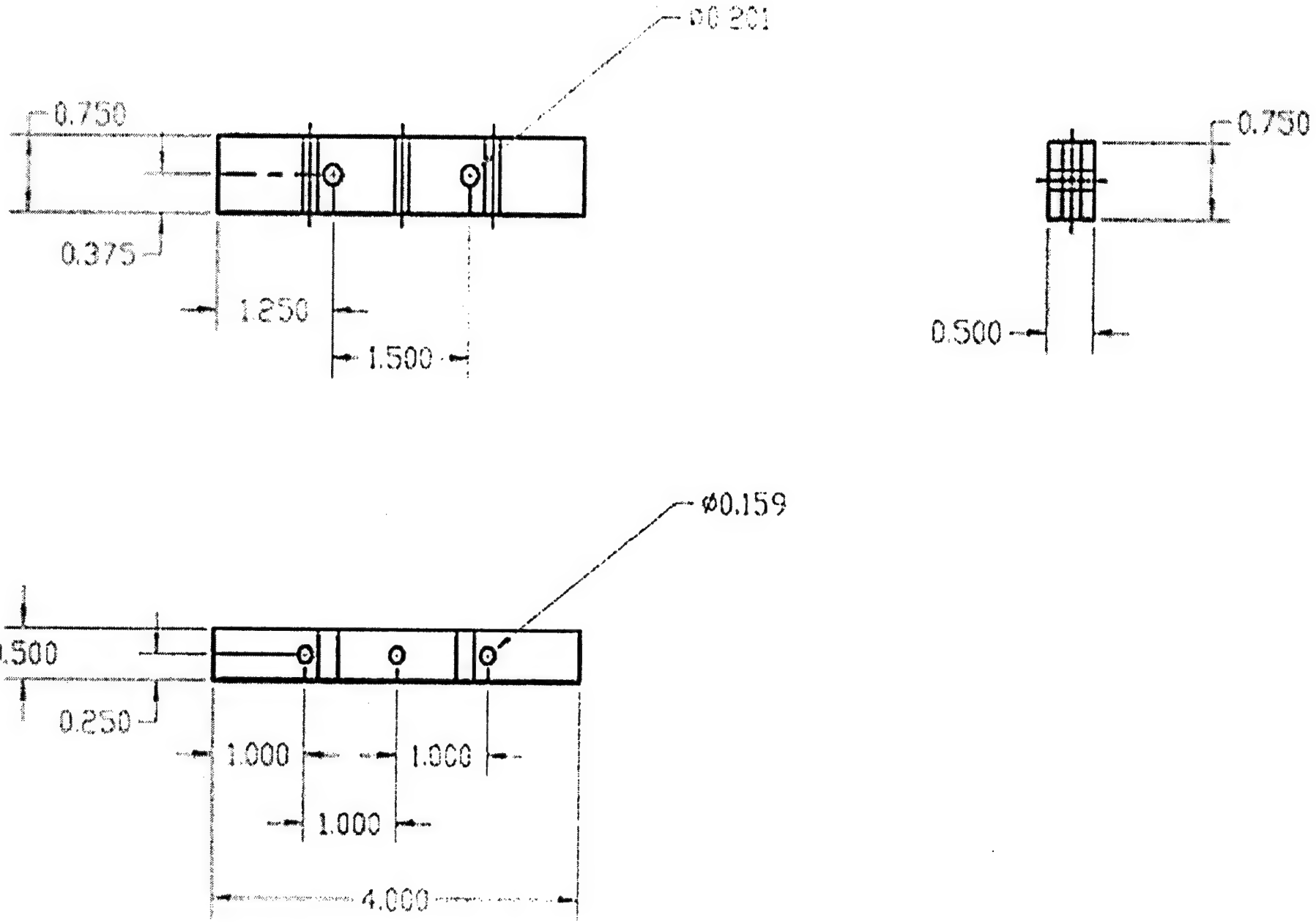


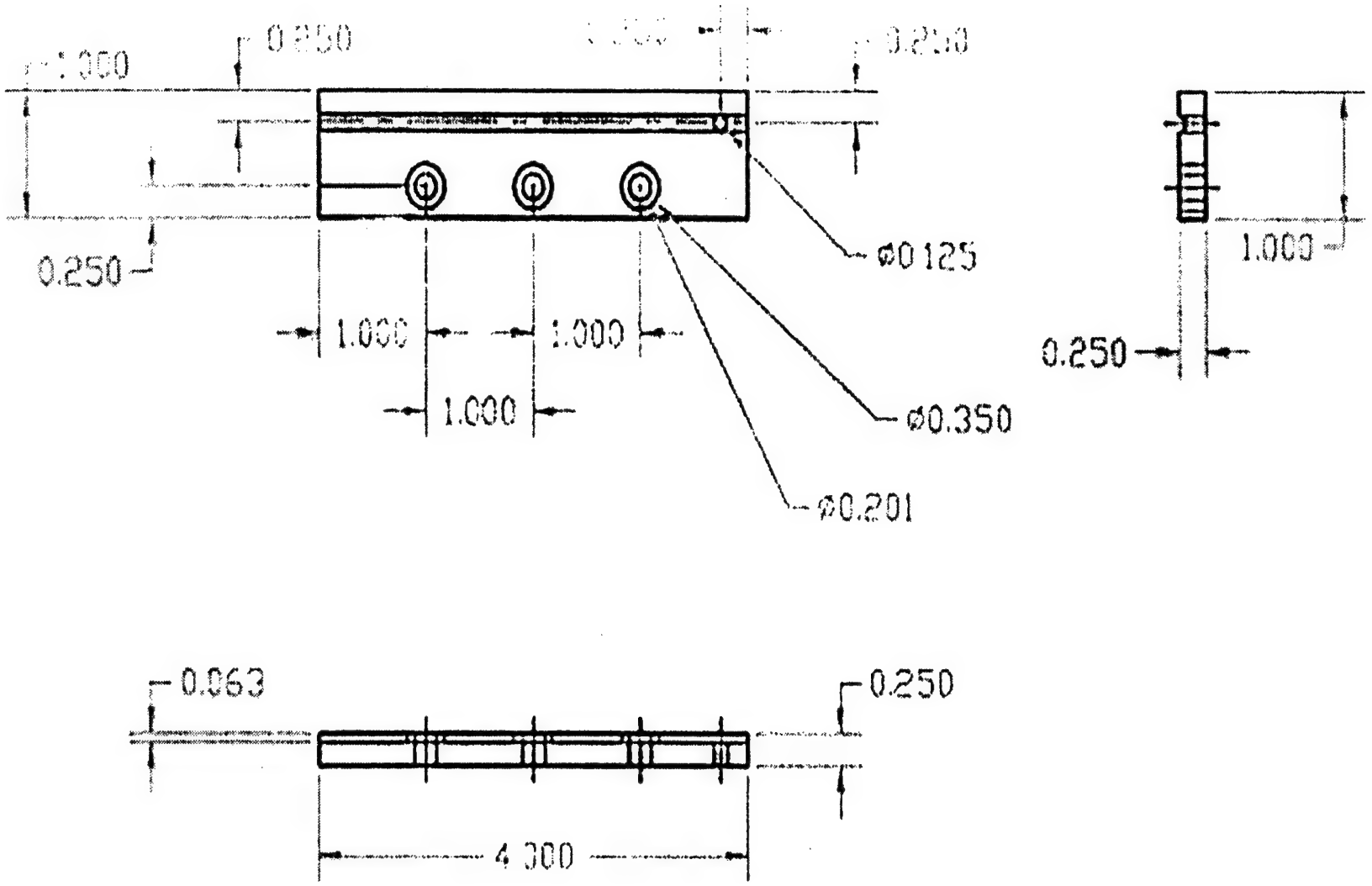
B. Mechanical Drawings Experiment #1--AutoCAD

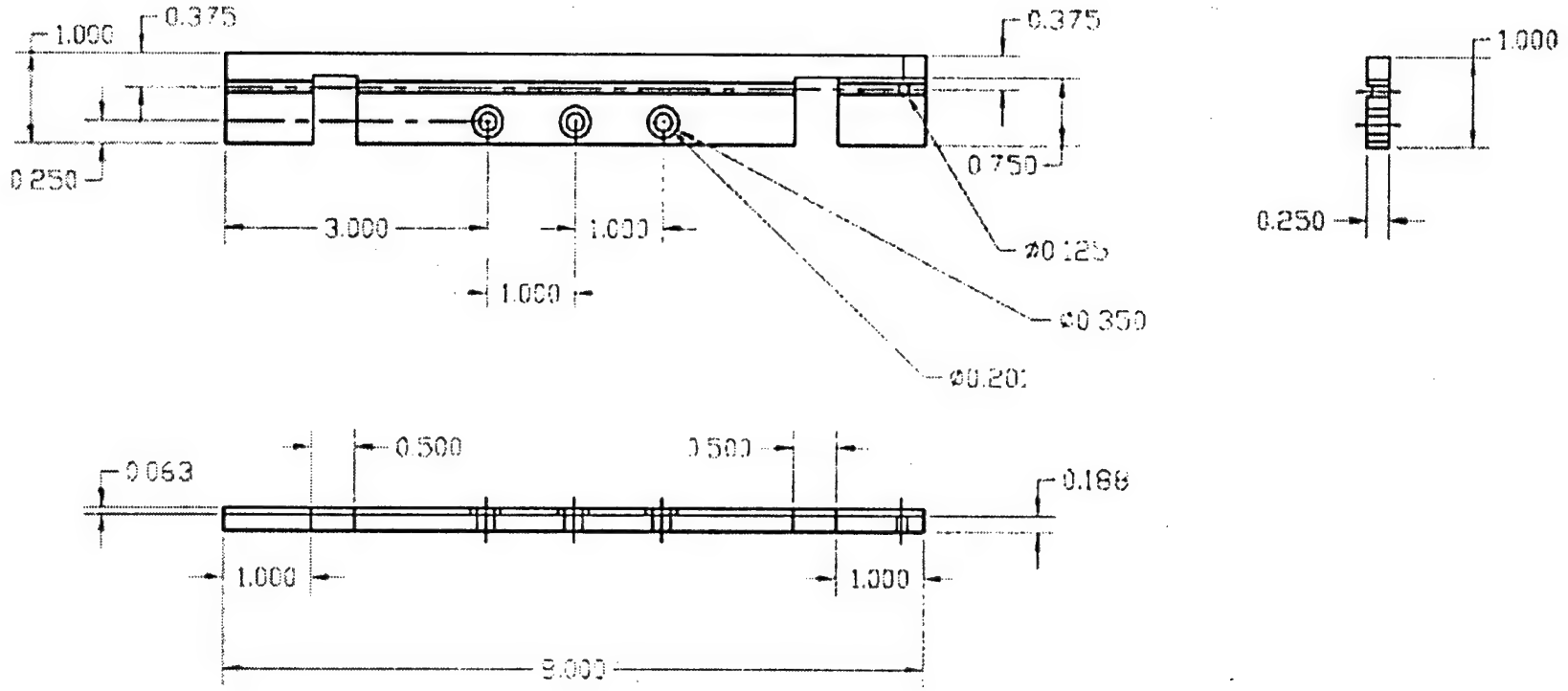


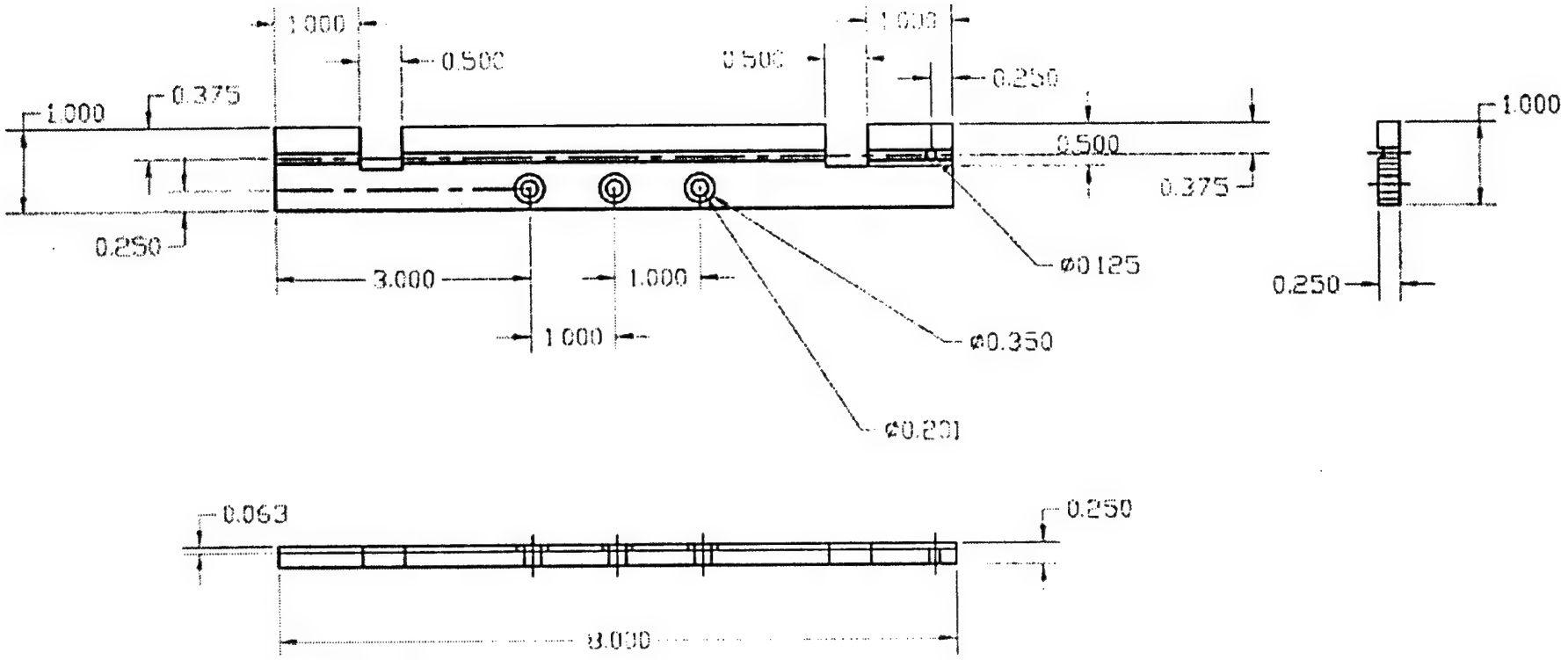



0.500
ø0.400
0.750
0.250
0.500
ø0.159
1.500
0.250
0.250
1.500

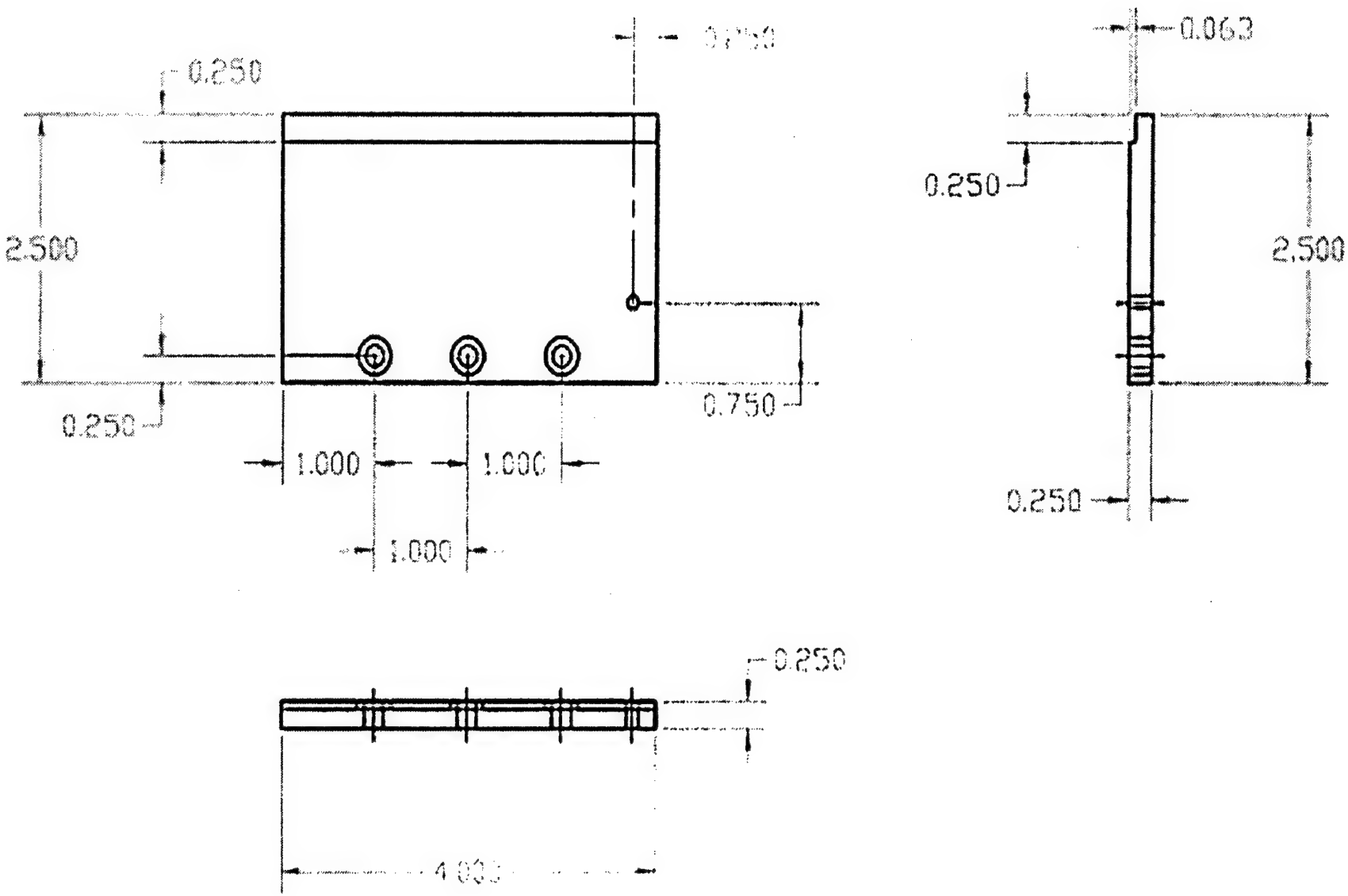

Ø0.201
0.750
0.375
1.250
1.500
0.750
0.500
Ø0.159
0.500
0.250
1.000
1.000
1.000
4.000

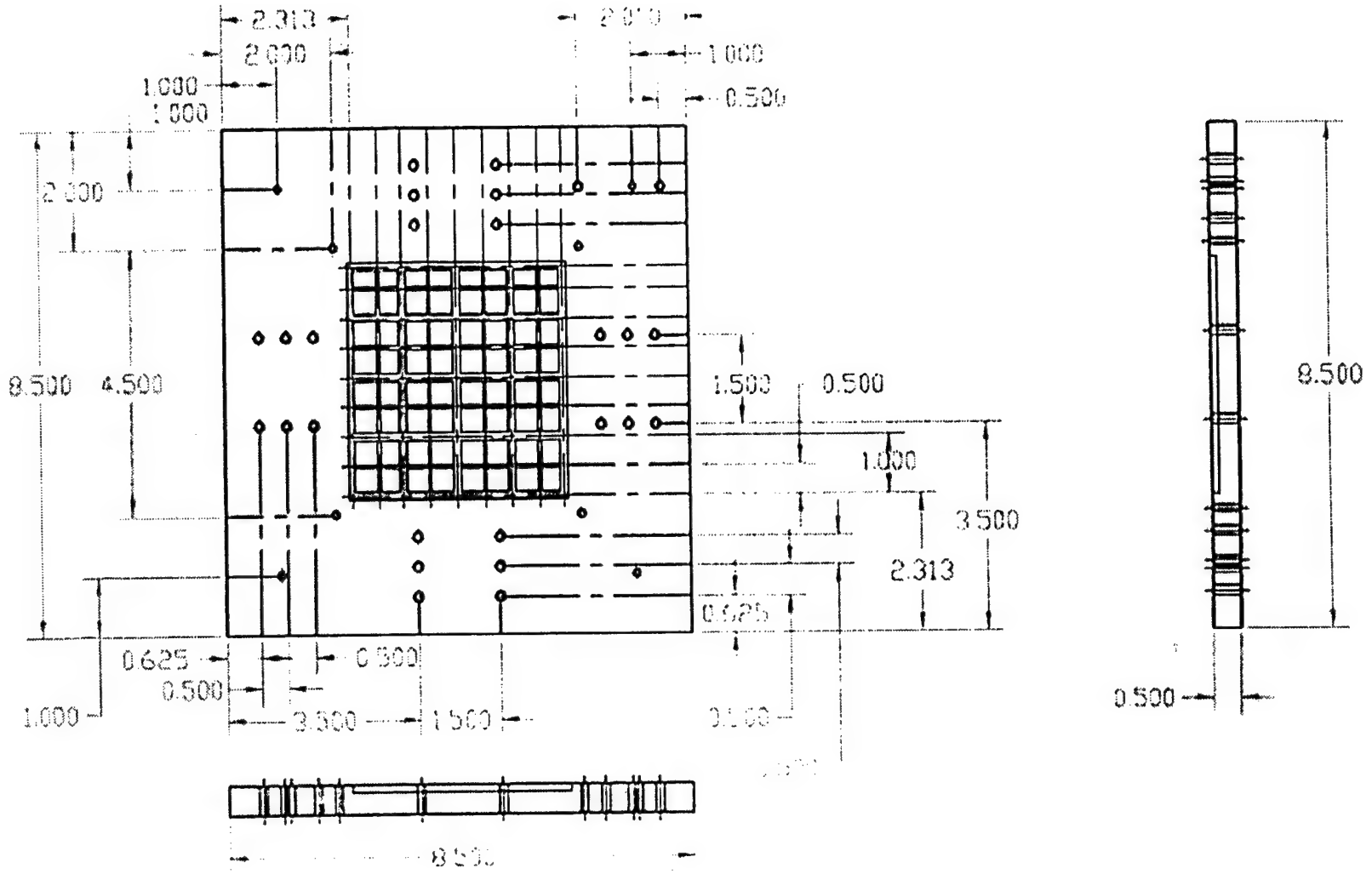

1.000
0.250
0.250
0.250
ø0.125
1.000
1.000
1.000
ø0.350
ø0.201
1.000
0.250
0.063
0.250
4.000

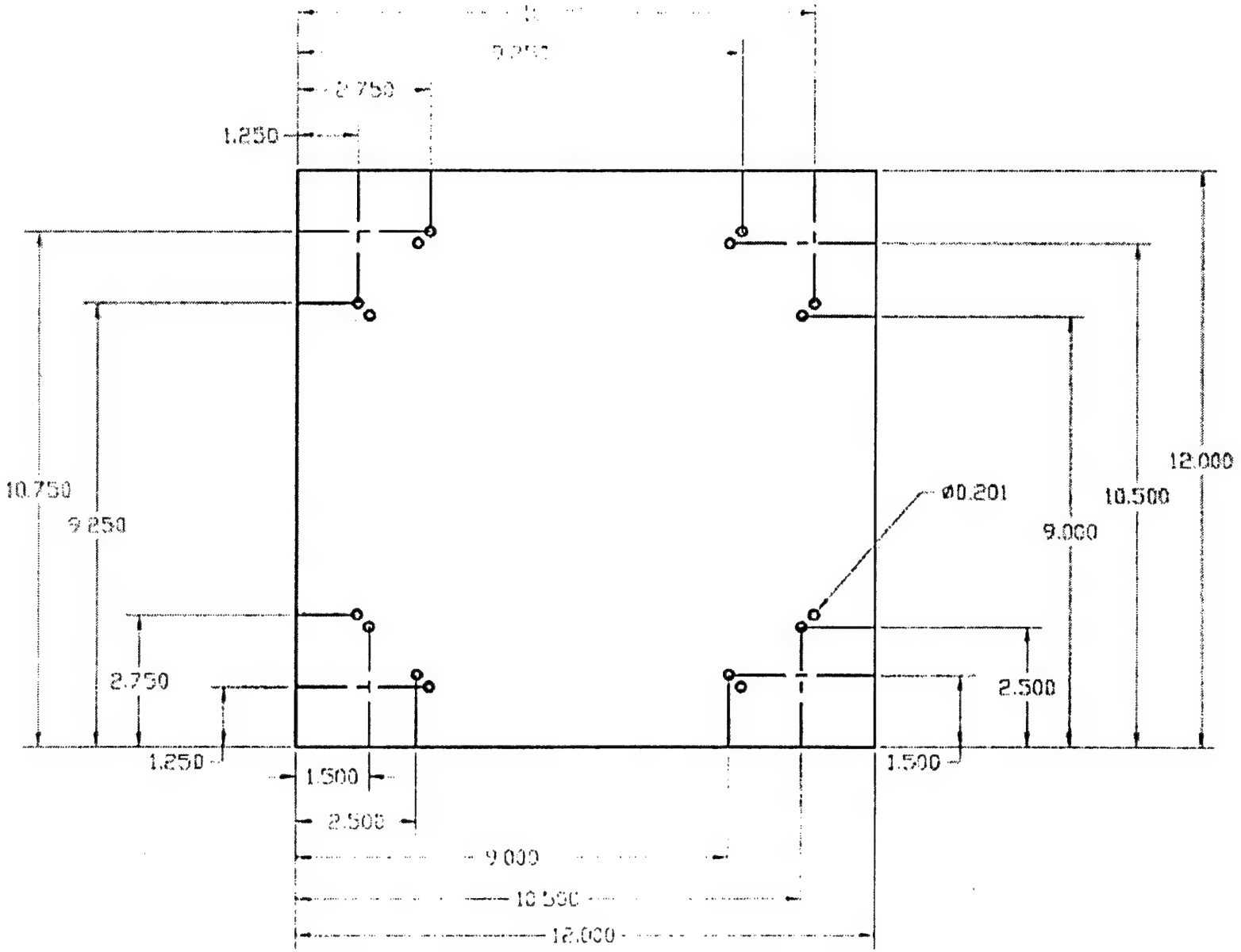

1.000
0.375
0.375
0.250
0.750
3.000
1.000
1.000
ø0.125
ø0.350
ø0.200
1.000
0.250
0.063
0.500
0.500
0.188
1.000
1.000
9.000

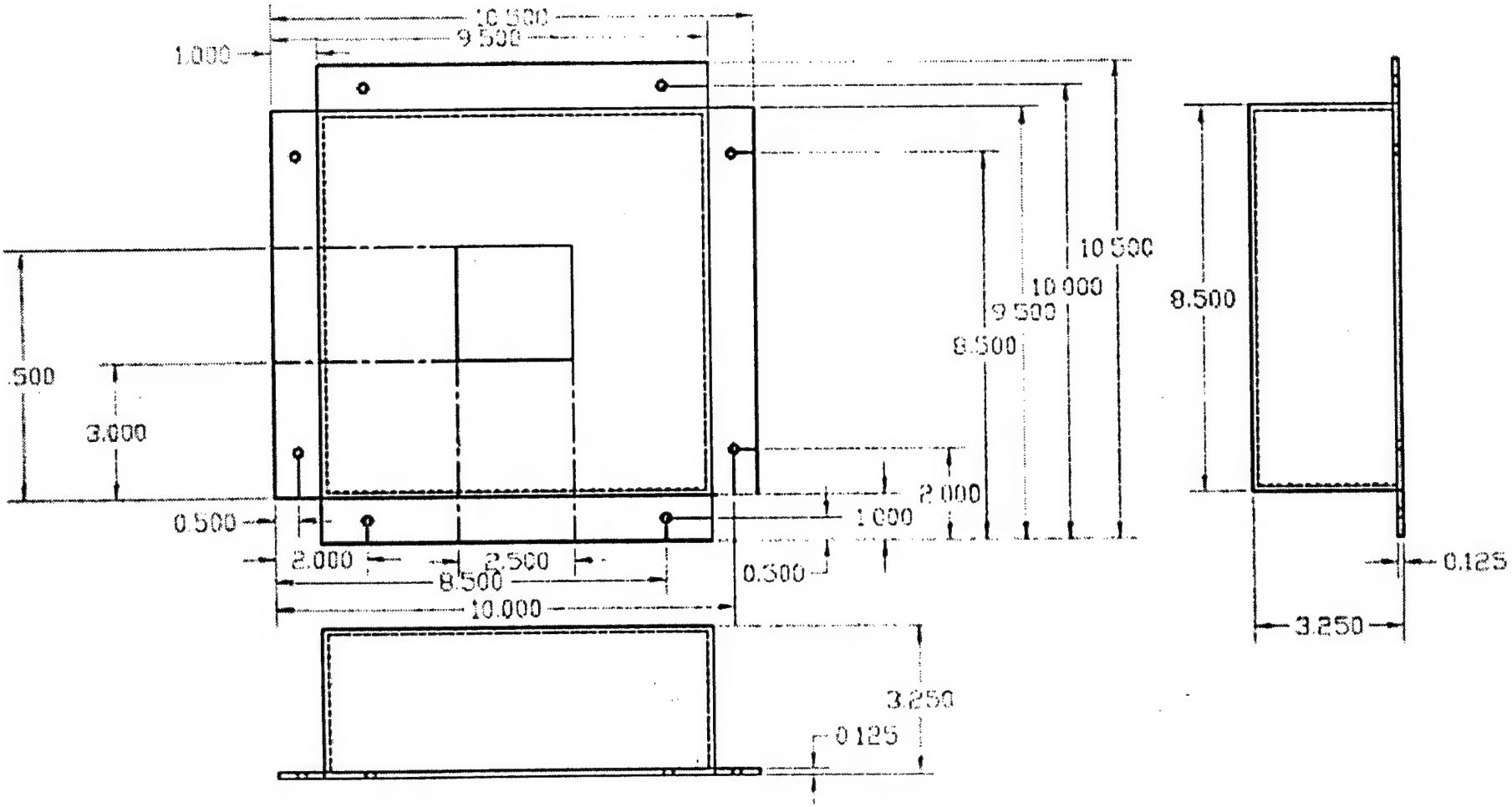

1.000
0.500
0.500
1.000
0.250
1.000
0.375
0.500
0.375
1.000
0.250
3.000
1.000
1.000
ø0.125
ø0.350
ø0.201
0.250

0.063
0.250
8.000

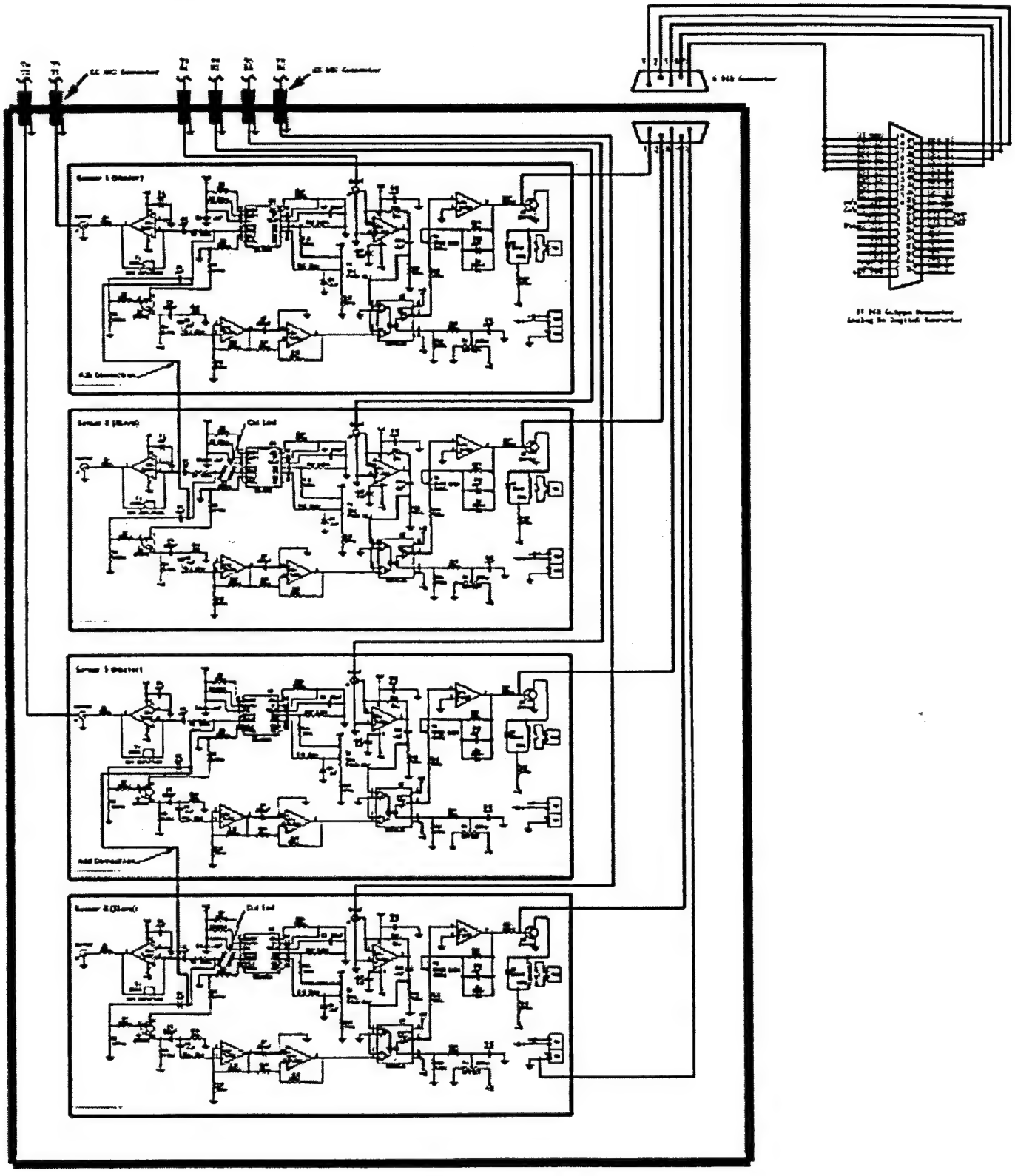

0.250
2.500
0.250
0.750
1.000
1.000
1.000
0.063
0.250
2.500
0.250
0.250


2.313
2.000
1.000
1.000
1.000
0.500
2.000
8.500
4.500
1.500
0.500
1.000
3.500
2.313
0.625
0.500
0.500
1.000
3.500
1.500
8.500
8.500
0.500

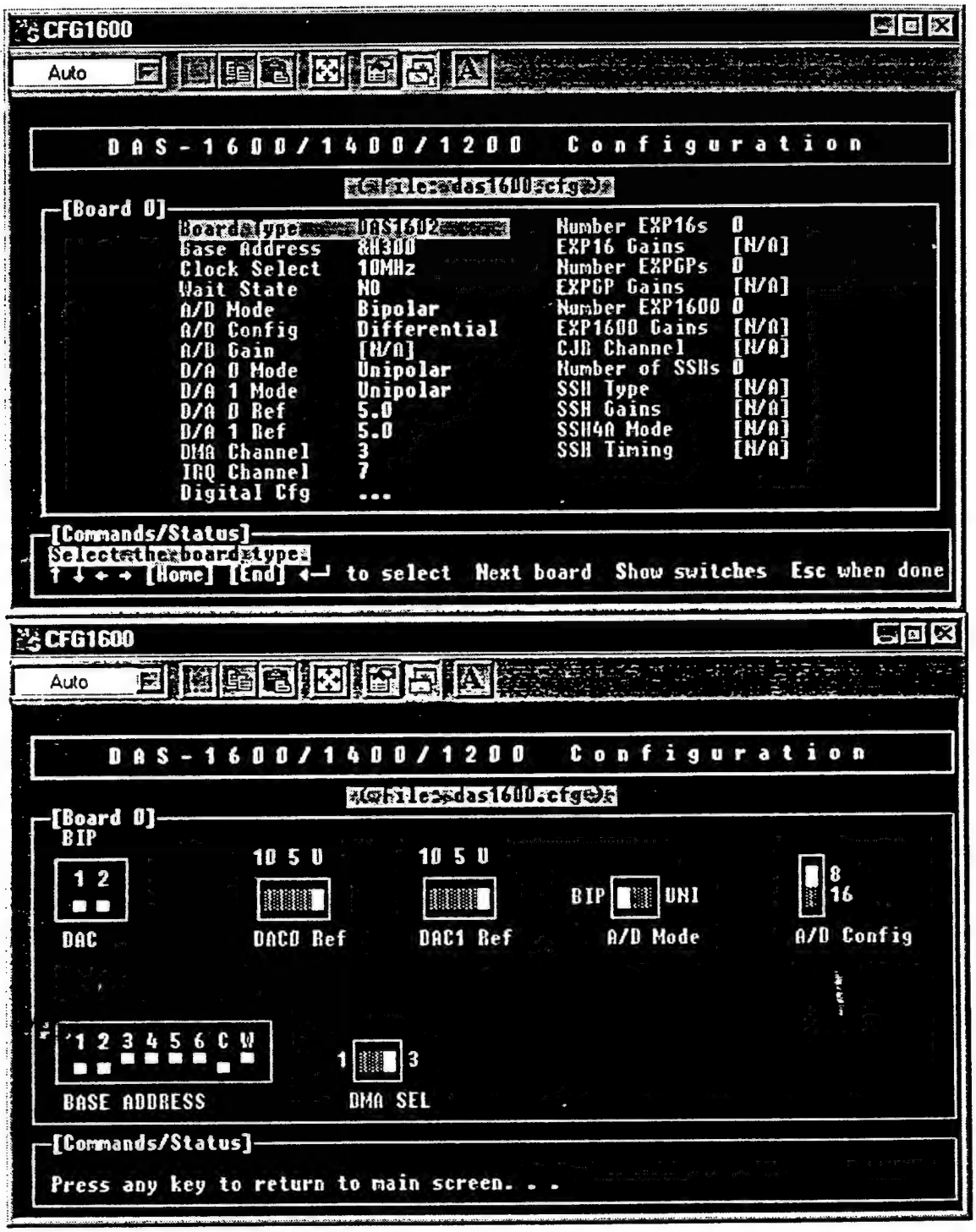

2.750
1.250
10.750
9.250
2.750
1.250
1.500
2.500
9.000
10.500
12.000
Ø0.201
10.500
9.000
12.000
2.500
1.500


10.500
9.500
1.000
10.500
10.000
9.500
8.500
.500
3.000
2.000
1.000
0.500
2.000
2.500
8.500
10.000
0.300
3.250
0.125
8.500
0.125
3.250

C. Circuit Schematic Experiment #1



D. Keithley Data Acquisition Board DAS-1602



E. ASO-1600 C functions

The K_OpenDriver() Statement

The syntax for the open driver statement is as follows:

K_OpenDriver(char * *BoardName*, char * *CfgFile*, *&hDrv1600*);

BoardName is the name of the board for future reference when multiple boards are used.

CfgFile is the driver configuration file.

&hDrv1600 is the memory address of the driver handle for storing the driver handle for referencing the analog to digital board.

This function initializes the hardware and software for the DAS-1600 analog to digital acquisition board. In the case of an error while initializing the board, the function returns TRUE, otherwise the function returns FALSE and the board is ready.

Example

```
// Initialize the hardware and software
if(( nErr = K_OpenDriver( "DAS1600", "das1600.CFG", &hDrv1600)) != 0)
        {
                putchar(7);
                printf("Error %X during K_OpenDriver", nErr);
                exit(nErr);
        }
```

K_GetDevHandle()

The syntax for the get device handle statement is as follows:

K_GetDevHandle(DWORD *hDrv1600*, int *BoardNumber*, *&hDev1600*);

hDrv1600 is the driver handle for accessing the analog to digital board as set by the K_OpenDriver statement.

BoardNumber is the number of the board for future reference when multiple boards are used.

&hDev1600 is the memory address of the device handle for storing the device handle for referencing the analog to digital board.

This function establishes communication with the driver through a device handle. On return from the function, the hDev1600 contains the handle associated with the board identified by *BoardNumber* and *hDrv1600*. In the case of an error while initializing the board, the function returns TRUE, otherwise the function returns FALSE and the board is ready.

Example

```
// Establish communication with the driver through a device handle
if(( nErr = K_GetDevHandle(hDrv1600, 0, &hDev1600)) != 0)
      {
            putchar(7);
            printf("Error %X during K_GetDevHandle ", nErr);
            exit(1);
      }
```

K_ADRead()

The syntax for the read analog to digital statement is as follows:

K_ADRead(DWORD *hdev1600*, unsigned char *Channel*, unsigned char *GainCode*, void * *wADval*);

hdev1600 is the handle to acquisition the board.

Channel is the input channel number;
0, 1,..., 15 (Single-ended configuration)
0, 1,..., 7 (Differential configuration)

GainCode is the gain setting for the analog value being acquired:

Gain Code	1601 gain	1601 input range	1602 gain	1602 input range
0	1	+- 10 V Bipolar 0-10 V Unipolar	1	+- 10 V Bipolar 0-10 V Unipolar
1	10	+- 1 V Bipolar 0-1 V Unipolar	2	+- 5 V Bipolar 0-5 V Unipolar
2	100	+- 0.1 V Bipolar 0-0.1 V Unipolar	4	+- 2.5 V Bipolar 0-2.5 V Unipolar
3	500	+- 0.02 V Bipolar 0-0.02 V Unipolar	8	+- 1.25 V Bipolar 0-1.25 V Unipolar

wADval is the storage location of the acquired analog to digital value.

This function uses the board identified by *hdev1600* to perform a single analog to digital acquisition. The value is acquired on *Channel* and stored in *wADval*. *GainCode* specifies the gain to be applied to *Channel*. In the case of an error while initializing the board, the function returns TRUE, otherwise the function returns FALSE and the board is ready.

Example

```
// Read channel 0 at gain 1; store sample in Advalue
if((nErr = K_ADRead(hDev1600, a, 0, &wADval)) != 0)
      {
            putchar(7);
            printf("Error %X in K_ADRead operation.", nErr);
            exit(1);
      }
```

```
// Strip channel tag and display ADvalue
printf("A/D value from channel 0 is: %x\n", (wADval>>4)&0xfff);
```

K_CloseDriver()

The syntax for the close driver statement is as follows:

K_CloseDriver(DWORD hDrv1600);

hDrv1600 is the driver handle for accessing the analog to digital board as set by the K_OpenDriver statement.

This function closes the hardware and software for the DAS-1600 analog to digital acquisition board and releases all resources.

Example

```
// Close the Driver and Release All Resources
K_CloseDriver(hDrv1600);
```

F Complete Software Code for Experiment 1

The following are the data collection programs for experiment 1:

```
/**************************************************************
*****

Program:                    Testbed Code

File:                               testbed.h

Function:                   Header for all testbed code

Description:

Author:                             Brian Fitch (BF)

Environment:        Microsoft Visual C++ 5.0, 486/66 16mb RAM, Windows 95
                                    DOS 6.0.

Notes:                              For use with testbed.c, testbed1.c,
testbed2.c, and testbed3.c

Revisions:                  1.00        3/10/98   (BF)     Initial Release
**************************************************************
******/

#include <windows.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>

#define IDS_ERR_REGISTER_CLASS  1
#define IDS_ERR_CREATE_WINDOW   2
#define STARTBTN            101
#define STOPBTN             102
#define CLOSEBTN            118
#define CONTBOX             124
#define STARTCHANBOX        104
#define STOPCHANBOX         105
#define NUMSAMPLESBOX       108
#define SAMPLERATEBOX       110
#define DMASTATUSBOX        113
#define DMATRANSFERBOX      114
#define DATALISTBOX         119
#define ID_TIMER            1

char acString[128];  /* variable to load resource strings       */
char *szString = acString;

char *szAppName = "              "; // class name for the window

HWND hInst;
HWND hWndMain;

LONG FAR PASCAL WndProc(HWND, UINT, UINT, LONG);
int  nCwRegisterClasses(void);
void CwUnRegisterClasses(void);
void InitWindowFields( HWND hwnd );
void InitDASDevice(void) ;
void StartAcquiring(HWND hWnd) ;
void StopAcquiring(HWND hWnd) ;
void ProcessTimer(HWND hWnd);
void ShowData(HWND hWnd) ;
void ProcessError(short ErrNum) ;

■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■

/**************************************************************
*****

Program:                    Testbed Code

File:                               tbed0b.c

Function:                           main function listing

Description:            Reads 10 points in rapid succession at each point on the
                                    grid (1-25)

Author:                             Brian Fitch (BF)

Environment:        Microsoft Visual C++ 5.0, 486/66 16mb RAM, Windows 95
                                    DOS 6.0.

Notes:                              For use with Keithley DAS-1600
Board

Revisions:                  1.00        3/10/98   (BF)     Initial Release
**************************************************************
******/

#include <stdio.h>
#include "testbed.h"
#include "dasdecl.h"
//#include "das1600.h"

void clear_kb(void);
int initialize_text(void);
int check_ESC(void);
int check_Q(void);
int checker(void);
int get_values(void);
int output_text(void);
int close_driver(void);
int initialize_A2D(void);

int SampleNumber=1;                     // Sample position number (1-64)
int PosX=0;                                     // X-coordinate on grid
int PosY=0;                                     // Y-coordinate on grid
double Voltage[4];              // Converted Voltage reading
int Counts[4];                          // Counts reading from a2d card
int RunNumber=1;                // Run Number of data at a single point (1-10)

DWORD hDrv1600;                         // Driver Handle
DWORD hDev1600;                         // Device Handle
short nErr;                                     // Function return error flag
WORD wADval;                            // Storage for A/D value

FILE *fp;                                       // File open error flag
char filename[20];              // Temp value for file name
int check=0;                            // Function return interrupt flag


int main(void)
{
    clear_kb();

    // open a file for storing data
    puts("Enter a name for the file");
    gets(filename);
    if((fp=fopen(filename, "w")) == NULL)
    {
                fprintf(stderr, "Error opening file %s.", filename);
                exit(1);
    }

    initialize_text();
    fprintf(stdout, "\nText successfully initialized");

    initialize_A2D();
    fprintf(stdout, "\nPress ESC to begin gathering data");
    fprintf(stdout, "\nPress Q to Quit");
```

```
    for(;;)
    {
                    if((check = checker()) == 1)
                    {
                            fprintf(stdout, "\nGathering Sample %d (%d, %d)
", SampleNumber, PosX, PosY);
                            for(RunNumber=1; RunNumber<11;
RunNumber++)
                            {
                                    get_values();
                                    output_text();
                                    fprintf(stdout, ".");
                            }

                            SampleNumber++;
                            PosX++;

                            if(PosX==5)
                            {

                                    PosY++;
                                    PosX=0;
                            }

                            fprintf(stdout, " Done\a");

                            if(SampleNumber > 25)
                                    break;
                    }
                    if(check == 2)
                            break;
    }

    close_driver();

    return 0;
}

int initialize_text(void)
{
    fprintf(fp, "\nTest Number\tX Location\tY Location\tRun
Number\tV1\tV2\tV3\tV4\tC1\tC2\tC3\tC4\n");
    return 0;
}

int output_text(void)
{
    fprintf(fp, "\n%d\t%d\t%d\t%d\t%f\t%f\t%f\t%f\t%d\t%d\t%d\t%d",
SampleNumber, PosX, PosY, RunNumber, Voltage[0], Voltage[1], Voltage[2],
Voltage[3], Counts[0], Counts[1], Counts[2], Counts[3]);
    return 0;
}

int get_values(void)
{
    int a = 0;


    for (a=0; a < 4; a++)
    {

                    // Read channel 0 at gain 1; store sample in Advalue
                    if((nErr = K_ADRead(hDev1600, a, 0, &wADval)) != 0)
                    {
                            putchar(7);
                            printf("Error %X in K_ADRead operation.",
nErr);
                            exit(1);
                    }

                    // Strip channel tag and display ADvalue
                    // printf("A/D value from channel 0 is: %x\n",
(wADval>>4)&0xfff);

                    Counts[a] = (wADval>>4)&0xfff;
                    Voltage[a] = ((((double)Counts[a]) - 2048) * (20)) / (4096);
    }

    return 0;
}
```

```
int initialize_A2D(void)
{
    // Initialize the hardware and software
    if(( nErr = K_OpenDriver( "DAS1600", "das1600.CFG", &hDrv1600)) != 0)
    {
                    putchar(7);
                    printf("Error %X during K_OpenDriver", nErr);
                    exit(nErr);
    }

    // Establish communication with the driver through a device handle
    if(( nErr = K_GetDevHandle(hDrv1600, 0, &hDev1600)) != 0)
    {
                    putchar(7);
                    printf("Error %X during K_GetDevHandle ", nErr);
                    exit(1);
    }
    return 0;
}

int close_driver(void)
{
    // Close the Driver and Release All Resources
    K_CloseDriver(hDrv1600);

    return 0;
}

int check_ESC(void)
{
   int in;
   if (kbhit())
    {
      if ((in = getch()) == '\x1B')
                    return 1;
    }
   else
                    return 0;
}

int check_Q(void)
{
    int in;
    if (kbhit())
    {
                    if ((in = getch()) == '\x51')
                    return 1;
    }
    else
                    return 0;
}

int checker(void)
{
    int in;
    if (kbhit())
    {
                    if ((in=getch()) == '\x1B')
                            return 1;
                    if (in=='\x51')
                            return 2;
                    return 0;
    }
}

void clear_kb(void)
// clears stdin of any waiting characters.
{
    char junk[80];
    gets(junk);
}
```

```
/*****************************************************************
*****

Program:                    Testbed Code
```

```
File:                   tbed1b.c

Function:               main function listing

Description:            Reads 10 points in rapid succession at each point on the
                        grid (1-81)

Author:                 Brian Fitch (BF)

Environment:            Microsoft Visual C++ 5.0, 486/66 16mb RAM, Windows 95
                        DOS 6.0.

Notes:                  For use with Keithley DAS-1600
Board

Revisions:              1.00    3/10/98    (BF)    Initial Release
*************************************************************
******/


#include <stdio.h>
#include "testbed.h"
#include "dasdecl.h"
//#include "das1600.h"

void clear_kb(void);
int initialize_text(void);
int check_ESC(void);
int check_Q(void);
int checker(void);
int get_values(void);
int output_text(void);
int close_driver(void);
int initialize_A2D(void);

int SampleNumber=1;             // Sample position number (1-64)
int PosX=0;                     // X-coordinate on grid
int PosY=0;                     // Y-coordinate on grid
double Voltage[4];              // Converted Voltage reading
int Counts[4];                  // Counts reading from a2d card
int RunNumber=1;                // Run Number of data at a single point (1-10)

DWORD hDrv1600;                 // Driver Handle
DWORD hDev1600;                 // Device Handle
short nErr;                     // Function return error flag
WORD wADval;                    // Storage for A/D value

FILE *fp;                       // File open error flag
char filename[20];              // Temp value for file name
int check=0;                    // Function return interrupt flag


int main(void)
{
    clear_kb();

    // open a file for storing data
    puts("Enter a name for the file");
    gets(filename);
    if((fp=fopen(filename, "w")) == NULL)
    {
            fprintf(stderr, "Error opening file %s.", filename);
            exit(1);
    }

    initialize_text();
    fprintf(stdout, "\nText successfully initialized");

    initialize_A2D();
    fprintf(stdout, "\nPress ESC to begin gathering data");
    fprintf(stdout, "\nPress Q to Quit");

    for(;;)
    {
            if((check = checker()) == 1)
            {
                    fprintf(stdout, "\nGathering Sample %d (%d, %d)
", SampleNumber, PosX, PosY);
                    for(RunNumber=1; RunNumber<11;
RunNumber++)
                    {
                            get_values();
                            output_text();
                            fprintf(stdout, ".");
                    }

                    SampleNumber++;
                    PosX++;

                    if(PosX==9)
                    {

                            PosY++;
                            PosX=0;
                    }

                    fprintf(stdout, " Done\a");

                    if(SampleNumber > 81)
                            break;
            }
            if(check == 2)
                    break;
    }

    close_driver();

    return 0;
}

int initialize_text(void)
{
    fprintf(fp, "\nTest Number\tX Location\tY Location\tRun
Number\tV1\tV2\tV3\tV4\tC1\tC2\tC3\tC4\n");
    return 0;
}

int output_text(void)
{
    fprintf(fp, "\n%d\t%d\t%d\t%d\t%f\t%f\t%f\t%f\t%d\t%d\t%d\t%d",
SampleNumber, PosX, PosY, RunNumber, Voltage[0], Voltage[1], Voltage[2],
Voltage[3], Counts[0], Counts[1], Counts[2], Counts[3]);
    return 0;
}

int get_values(void)
{
    int a = 0;


    for (a=0; a < 4; a++)
    {

            // Read channel 0 at gain 1; store sample in Advalue
            if((nErr = K_ADRead(hDev1600, a, 0, &wADval)) != 0)
            {
                    putchar(7);
                    printf("Error %X in K_ADRead operation.",
nErr);
                    exit(1);
            }

            // Strip channel tag and display ADvalue
            // printf("A/D value from channel 0 is: %x\n",
(wADval>>4)&0xfff);

            Counts[a] = (wADval>>4)&0xfff;
            Voltage[a] = ((((double)Counts[a]) - 2048) * (20)) / (4096);
    }

    return 0;
}

int initialize_A2D(void)
{
    // Initialize the hardware and software
    if(( nErr = K_OpenDriver( "DAS1600", "das1600.CFG", &hDrv1600)) != 0)
    {
            putchar(7);
            printf("Error %X during K_OpenDriver", nErr);
            exit(nErr);
    }
```

```
    // Establish communication with the driver through a device handle
    if(( nErr = K_GetDevHandle(hDrv1600, 0, &hDev1600)) != 0)
    {
                    putchar(7);
                    printf("Error %X during K_GetDevHandle ", nErr);
                    exit(1);
    }
    return 0;
}

int close_driver(void)
{
    // Close the Driver and Release All Resources
    K_CloseDriver(hDrv1600);

    return 0;
}

int check_ESC(void)
{
  int in;
  if (kbhit())
    {
      if ((in = getch()) == '\x1B')
                    return 1;
    }
  else
                    return 0;
}

int check_Q(void)
{
    int in;
    if (kbhit())
    {
                    if ((in = getch()) == '\x51')
                    return 1;
    }
    else
                    return 0;
}

int checker(void)
{
    int in;
    if (kbhit())
    {
                    if ((in=getch()) == '\x1B')
                            return 1;
                    if (in=='\x51')
                            return 2;
                    return 0;
    }
}

void clear_kb(void)
'/ clears stdin of any waiting characters.
{
    char junk[80];
    gets(junk);
}
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

/*************************************************************************
*****

Program:                    Testbed Code

File:                               tbed2b.cpp

Function:                   main function listing

Description:          Reads 10 points in succession at each point on the
                                    grid (1-81) with a 1 second delay
between readings

Author:                             Brian Fitch (BF)

Environment:          Microsoft Visual C++ 5.0, 486/66 16mb RAM, Windows 95
                                              DOS 6.0.

Notes:                                        For use with Keithley DAS-1600
Board

Revisions:                    1.00        3/10/98     (BF)        Initial Release
*************************************************************************
******/

#include <afxwin.h>
#include <stdio.h>
#include <winbase.h>
#include "testbed.h"
#include "dasdecl.h"
//#include "das1600.h"

void clear_kb(void);
int initialize_text(void);
int check_ESC(void);
int check_Q(void);
int checker(void);
int get_values(void);
int output_text(void);
int close_driver(void);
int initialize_A2D(void);

int SampleNumber=1;                     // Sample position number (1-64)
int PosX=0;                                     // X-coordinate on grid
int PosY=0;                                     // Y-coordinate on grid
double Voltage[4];              // Converted Voltage reading
int Counts[4];                          // Counts reading from a2d card
int RunNumber=1;                // Run Number of data at a single point (1-10)

DWORD hDrv1600;                         // Driver Handle
DWORD hDev1600;                         // Device Handle
short nErr;                                     // Function return error flag
WORD wADval;                            // Storage for A/D value

FILE *fp;                                       // File open error flag
char filename[20];              // Temp value for file name
int check=0;                            // Function return interrupt flag


int main(void)
{
    clear_kb();

    // open a file for storing data
    puts("Enter a name for the file");
    gets(filename);
    if((fp=fopen(filename, "w")) == NULL)
    {
                    fprintf(stderr, "Error opening file %s.", filename);
                    exit(1);
    }

    initialize_text();
    fprintf(stdout, "\nText successfully initialized");

    initialize_A2D();
    fprintf(stdout, "\nPress ESC to begin gathering data");
    fprintf(stdout, "\nPress Q to Quit");

    for(;;)
    {
                    if((check = checker()) == 1)
                    {
                            fprintf(stdout, "\nGathering Sample %d (%d, %d)
", SampleNumber, PosX, PosY);
                            for(RunNumber=1; RunNumber<11;
RunNumber++)
                            {
                                    get_values();
                                    output_text();
                                    fprintf(stdout, ".");
                                    Sleep(1000);
                            }

                            SampleNumber++;
                            PosX++;
```

```
                        if(PosX==9)
                        {

                                PosY++;
                                PosX=0;
                        }

                        fprintf(stdout, " Done\a");

                        if(SampleNumber > 81)
                                break;
                }
                if(check == 2)
                        break;
        }

        close_driver();

        return 0;
}

int initialize_text(void)
{
        fprintf(fp, "\nTest Number\tX Location\tY Location\tRun
Number\tV1\tV2\tV3\tV4\tC1\tC2\tC3\tC4\n");
        return 0;
}

int output_text(void)
{
        fprintf(fp, "\n%d\t%d\t%d\t%d\t%f\t%f\t%f\t%f\t%d\t%d\t%d\t%d",
SampleNumber, PosX, PosY, RunNumber, Voltage[0], Voltage[1], Voltage[2],
Voltage[3], Counts[0], Counts[1], Counts[2], Counts[3]);
        return 0;
}

int get_values(void)
{
        int a = 0;

        for (a=0; a < 4; a++)
        {

                // Read channel 0 at gain 1; store sample in Advalue
                if((nErr = K_ADRead(hDev1600, a, 0, &wADval)) != 0)
                {
                        putchar(7);
                        printf("Error %X in K_ADRead operation.",
nErr);
                        exit(1);
                }

                // Strip channel tag and display ADvalue
                // printf("A/D value from channel 0 is: %x\n",
(wADval>>4)&0xfff);

                Counts[a] = (wADval>>4)&0xfff;
                Voltage[a] = ((double)Counts[a]) - 2048) * (20)) / (4096);
        }

        return 0;
}

int initialize_A2D(void)
{
        // Initialize the hardware and software
        if(( nErr = K_OpenDriver( "DAS1600", "das1600.CFG", &hDrv1600)) != 0)
        {
                putchar(7);
                printf("Error %X during K_OpenDriver", nErr);
                exit(nErr);
        }

        // Establish communication with the driver through a device handle
        if(( nErr = K_GetDevHandle(hDrv1600, 0, &hDev1600)) != 0)
        {
                putchar(7);
                printf("Error %X during K_GetDevHandle ", nErr);
                exit(1);
        }
        return 0;
}

int close_driver(void)
{
        // Close the Driver and Release All Resources
        K_CloseDriver(hDrv1600);

        return 0;
}

int check_ESC(void)
{
  int in;
  if (kbhit())
    {
      if ((in = getch()) == '\x1B')
                return 1;
    }
  else
                return 0;
        return 0;
}

int check_Q(void)
{
        int in;
        if (kbhit())
        {
                if ((in = getch()) == '\x51')
                return 1;
        }
        else
                return 0;
        return 0;
}

int checker(void)
{
        int in;
        if (kbhit())
        {
                if ((in=getch()) == '\x1B')
                        return 1;
                if (in=='\x51')
                        return 2;
                return 0;
        }
        return 0;
}

void clear_kb(void)
// clears stdin of any waiting characters.
{
        char junk[80];
        gets(junk);
}
```

```
/*****************************************************************
*****

Program:                Testbed Code

File:                   tbed3b.c

Function:               main function listing

Description:            Reads a single set of voltages at each point on the
                        grid (1-81). Repeats 10 through

Author:                 Brian Fitch (BF)

Environment:            Microsoft Visual C++ 5.0, 486/66 16mb RAM, Windows 95
                        DOS 6.0.

Notes:                  For use with Keithley DAS-1600
Board
```

```
Revisions:                    1.00      3/10/98   (BF)      Initial Release
*****************************************************************
******/

#include <stdio.h>
#include "testbed.h"
#include "dasdecl.h"
//#include "das1600.h"

void clear_kb(void);
int initialize_text(void);
int check_ESC(void);
int check_Q(void);
int checker(void);
int get_values(void);
int get_values2(void);
int output_text(void);
int output_text2(void);
int close_driver(void);
int initialize_A2D(void);

int SampleNumber=1;                    // Sample position number (1-64)
int PosX=0;                                    // X-coordinate on grid
int PosY=0;                                    // Y-coordinate on grid
double Voltage[4];              // Converted Voltage reading
double Voltage2[82][11][4];
int Counts[4];                         // Counts reading from a2d card
int Counts2[82][11][4];
int RunNumber=1;                // Run Number of data at a single point (1-10)

DWORD hDrv1600;                        // Driver Handle
DWORD hDev1600;                        // Device Handle
short nErr;                                    // Function return error flag
WORD wADval;                           // Storage for A/D value

FILE *fp;                                      // File open error flag
char filename[20];              // Temp value for file name
int check=0;                           // Function return interrupt flag


int main(void)
{
    clear_kb();

    // open a file for storing data
    puts("Enter a name for the file");
    gets(filename);
    if((fp=fopen(filename, "w")) == NULL)
    {
                fprintf(stderr, "Error opening file %s.", filename);
                exit(1);
    }

    initialize_text();
    fprintf(stdout, "\nText successfully initialized");

    initialize_A2D();
    fprintf(stdout, "\nPress ESC to begin gathering data");
    fprintf(stdout, "\nPress Q to Quit");

    for(;;)
    {
                if((check = checker()) == 1)
                {
                        fprintf(stdout, "\nGathering Sample %d.%d (%d,
%d) ", SampleNumber, RunNumber, PosX, PosY);
                        get_values2();

                        SampleNumber++;
                        PosX++;

                        if(PosX==9)
                        {

                                PosY++;
                                if(PosY==9)
                                        PosY=0;
                                PosX=0;
                        }

                        fprintf(stdout, "... Done\a");

                        if((SampleNumber > 81) && (RunNumber ==
10))
                                break;
                        if(SampleNumber > 81)
                        {
                                RunNumber++;
                                SampleNumber=1;
                        }
                }
                if(check == 2)
                        break;
    }

    close_driver();
    output_text2();

    return 0;
}

int initialize_text(void)
{
    fprintf(fp, "\nTest Number\tX Location\tY Location\tRun
Number\tV1\tV2\tV3\tV4\tC1\tC2\tC3\tC4\n");
    return 0;
}

int output_text(void)
{
    fprintf(fp, "\n%d\t%d\t%d\t%d\t%f\t%f\t%f\t%d\t%d\t%d\t%d",
SampleNumber, PosX, PosY, RunNumber, Voltage[0], Voltage[1], Voltage[2],
Voltage[3], Counts[0], Counts[1], Counts[2], Counts[3]);
    return 0;
}

int output_text2(void)
{
    int SN=1, RN=1;

    PosX=PosY=0;

    for (SN=1; SN<82; SN++)
    {
                for (RN=1; RN<11; RN++)
                {
                        fprintf(fp,
"\n%d\t%d\t%d\t%d\t%f\t%f\t%f\t%d\t%d\t%d\t%d", SN, PosX, PosY, RN,
Voltage2[SN][RN][0], Voltage2[SN][RN][1], Voltage2[SN][RN][2],
Voltage2[SN][RN][3], Counts2[SN][RN][0], Counts2[SN][RN][1],
Counts2[SN][RN][2], Counts2[SN][RN][3]);
                }

                PosX++;

                if(PosX==9)
                {
                        PosY++;
                        if(PosY==9)
                                PosY=0;
                        PosX=0;
                }
    }

    return 0;
}

int get_values(void)
{
    int a = 0;

    for (a=0; a < 4; a++)
    {

                // Read channel 0 at gain 1; store sample in Advalue
                if((nErr = K_ADRead(hDev1600, a, 0, &wADval)) != 0)
                {
                        putchar(7);
                        printf("Error %X in K_ADRead operation.",
nErr);
                        exit(1);
                }
```

```
                    // Strip channel tag and display ADvalue
                    // printf("A/D value from channel 0 is: %x\n",
(wADval>>4)&0xfff);

                    Counts[a] = (wADval>>4)&0xfff;
                    Voltage[a] = ((((double)Counts[a]) - 2048) * (20)) / (4096);
    }

    return 0;
}

int get_values2(void)
{
    int a = 0;

    for (a=0; a < 4; a++)
    {

                    // Read channel 0 at gain 1; store sample in Advalue
                    if((nErr = K_ADRead(hDev1600, a, 0, &wADval)) != 0)
                    {
                            putchar(7);
                            printf("Error %X in K_ADRead operation.",
nErr);
                            exit(1);
                    }

                    // Strip channel tag and display ADvalue
                    // printf("A/D value from channel 0 is: %x\n",
(wADval>>4)&0xfff);

                    Counts2[SampleNumber][RunNumber][a] =
(wADval>>4)&0xfff;
                    Voltage2[SampleNumber][RunNumber][a] =
((((double)Counts2[SampleNumber][RunNumber][a]) - 2048) * (20)) / (4096);
    }

    return 0;

int initialize_A2D(void)
{
    // Initialize the hardware and software
    if(( nErr = K_OpenDriver( "DAS1600", "das1600.CFG", &hDrv1600)) != 0)
    {
                    putchar(7);
                    printf("Error %X during K_OpenDriver", nErr);
                    exit(nErr);
    }

    // Establish communication with the driver through a device handle
    if(( nErr = K_GetDevHandle(hDrv1600, 0, &hDev1600)) != 0)
    {
                    putchar(7);
                    printf("Error %X during K_GetDevHandle ", nErr);
                    exit(1);
    }
    return 0;
}

int close_driver(void)
{
    // Close the Driver and Release All Resources
    K_CloseDriver(hDrv1600);

    return 0;
}

int check_ESC(void)
{
  int in;
  if (kbhit())
    {
      if ((in = getch()) == '\x1B')
                    return 1;
    }
  else
                    return 0;
}

int check_Q(void)
{
    int in;
    if (kbhit())
    {
                    if ((in = getch()) == '\x51')
                    return 1;
    }
    else
                    return 0;
}

int checker(void)
{
    int in;
    if (kbhit())
    {
                    if ((in=getch()) == '\x1B')
                            return 1;
                    if (in=='\x51')
                            return 2;
                    return 0;
    }
}

void clear_kb(void)
// clears stdin of any waiting characters.
{
    char junk[80];
    gets(junk);
}
```

```
/*************************************************************************
*****


Program:                Testbed Code


File:                           Height.c


Function:               main function listing


Description:    Reads 10 points in rapid succession at each point on the
                                grid (1-25) at 4 different heights (0,
.25, .5, .75, 1)

Author:                         Brian Fitch (BF)

Environment:    Microsoft Visual C++ 5.0, 486/66 16mb RAM, Windows 95
                                DOS 6.0.

Notes:                          For use with Keithley DAS-1600
Board

Revisions:              1.00    3/10/98    (BF)    Initial Release
*************************************************************************
******/


#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "testbed.h"
#include "dasdecl.h"
//#include "das1600.h"

void clear_kb(void);
int initialize_text(void);
int check_ESC(void);
int check_Q(void);
```

```
int checker(void);
int get_values(void);
int output_text(void);

int SampleNumber=1;
int PosX=0;
int PosY=0;
double Voltage[4];
int Counts[4];
int RunNumber=1;
double height=0.0;

DWORD hDrv1600;                 // Driver Handle
DWORD hDev1600;                 // Device Handle
short nErr;                             // Function return error flag
WORD wADval;                    // Storage for A/D value

FILE *fp;
char filename[20];
int check=0;


int main(void)
{
    clear_kb();

    puts("Enter a name for the file");
    gets(filename);
    if((fp=fopen(filename, "w")) == NULL)
    {
                fprintf(stderr, "Error opening file %s.", filename);
                exit(1);
    }

    initialize_text();
    fprintf(stdout, "\nText successfully initialized");
    fprintf(stdout, "\nPress ESC to begin gathering data");

    for(;;)
    {
                if((check = checker()) == 1)
                {
                        fprintf(stdout, "\nGathering Sample %d (%d, %d,
%f) ", SampleNumber, PosX, PosY, height);
                        for(RunNumber=1; RunNumber<11;
RunNumber++)
                        {
                                get_values();
                                output_text();
                                fprintf(stdout, ".");
                        }

                        SampleNumber++;
                        PosX++;

                        if(PosX==5)
                        {

                                PosY++;
                                PosX=0;
                                if(PosY == 5)
                                        PosY=0;
                        }

                        fprintf(stdout, " Done\a");

                        if((SampleNumber > 25) & (height == 1.0))
                                break;

                        if(SampleNumber > 25)
                        {
                                height = (height + .25);
                                SampleNumber=1;
                        }

                }
                if(check == 2)
                        break;
    }

    return 0;
}

int initialize_text(void)
{
    fprintf(fp, "\nTest Number\tX Location\tY Location\tZ Location\tRun
Number\tV1\tV2\tV3\tV4\tC1\tC2\tC3\tC4\n");
    return 0;
}

int output_text(void)
{
    fprintf(fp, "\n%d\t%d\t%d\t%f\t%d\t%f\t%f\t%f\t%f\t%d\t%d\t%d\t%d",
SampleNumber, PosX, PosY, height, RunNumber, Voltage[0], Voltage[1],
Voltage[2], Voltage[3], Counts[0], Counts[1], Counts[2], Counts[3]);
    return 0;
}

int get_values(void)
{
    int a = 0;

    // Initialize the hardware and software
    if(( nErr = K_OpenDriver( "DAS1600", "das1600.CFG", &hDrv1600)) != 0)
    {
                putchar(7);
                printf("Error %X during K_OpenDriver", nErr);
                exit(nErr);
    }

    // Establish communication with the driver through a device handle
    if(( nErr = K_GetDevHandle(hDrv1600, 0, &hDev1600)) != 0)
    {
                putchar(7);
                printf("Error %X during K_GetDevHandle ", nErr);
                exit(1);
    }

    for (a=0; a < 4; a++)
    {

                // Read channel 0 at gain 1; store sample in Advalue
                if((nErr = K_ADRead(hDev1600, a, 0, &wADval)) != 0)
                {
                        putchar(7);
                        printf("Error %X in K_ADRead operation.",
nErr);
                        exit(1);
                }

                // Strip channel tag and display ADvalue
                // printf("A/D value from channel 0 is: %x\n",
(wADval>>4)&0xfff);

                Counts[a] = (wADval>>4)&0xfff;
                Voltage[a] = ((((double)Counts[a]) - 2048) * (20)) / (4096);
    }

    // Close the Driver and Release All Resources
    K_CloseDriver(hDrv1600);

    return 0;
}

int check_ESC(void)
{
  int in;
  if (kbhit())
    {
      if ((in = getch()) == '\x1B')
                return 1;
    }
  else
                return 0;
}

int check_Q(void)
{
    int in;
    if (kbhit())
    {
                if ((in = getch()) == '\x51')
```

```
                return 1;
    }
    else
                return 0;
}

int checker(void)
{
    int in;
    if (kbhit())
    {
                if ((in=getch()) == '\x1B')
                        return 1;
                if (in=='\x51')
                        return 2;
                return 0;
    }
}

void clear_kb(void)
// clears stdin of any waiting characters.
{
    char junk[80];
    gets(junk);
}
```

G Complete Code for thesis

The following are the two library files for the finger walker interface: finger.cpp and finger.h.

```
/*************************************************************
******

Program:        Testbed Code

File:                   testbed.h

Function:       Header for all testbed code

Description:

Author:                 Brian Fitch (BF)

Environment:    Microsoft Visual C++ 5.0, 486/66 16mb RAM, Windows 95
                        DOS 6.0.

Notes:                  For use with testbed.c, testbed1.c, testbed2.c, and
testbed3.c

Revisions:      1.00    3/10/98   (BF)     Initial Release
*************************************************************
******/

#include "stdafx.h"
#include <stdio.h>
#include <conio.h>
//#include <bios.h>
#include <math.h>
#include <stdlib.h>
#include <windows.h>
#include <string.h>
#include "finger.h"

//DAS-1600/1400 driver include file
#include "dasdecl.h"


DWORD hDrv1600;                 // Driver Handle
DWORD hDev1600;                 // Device Handle
short nErr;                     // Function return error flag
WORD wADval;                    // Storage for A/D value

int error = 0;                  // error = 0 :: no errors
// error = 1 :: system error

int Counts[4];
double data[4];                 /*  data[0] == x1

    data[1] == y1

    data[2] == x2

    data[3] == y2
                                                        */
double coords[4];

double old_coords[SAVE][4];

struct coord first, second, old_first={2,2}, old_second={2,2};          //
(2,2) is the center of a 4 by 4 pad

// lookup[X][Y]
struct coord lookup_1[ROW][COL]={
{ {0,0}, {0,0.5}, {0,1}, {0,1.5}, {0,2}, {0,2.5}, {0,3}, {0,3.5}, {0,4}},
{ {0.5,0}, {0.5,0.5}, {0.5,1}, {0.5,1.5}, {0.5,2}, {0.5,2.5}, {0.5,3}, {0.5,3.5},
{0.5,4}},
{ {1,0}, {1,0.5}, {1,1}, {1,1.5}, {1,2}, {1,2.5}, {1,3}, {1,3.5}, {1,4}},
{ {1.5,0}, {1.5,0.5}, {1.5,1}, {1.5,1.5}, {1.5,2}, {1.5,2.5}, {1.5,3}, {1.5,3.5},
{1.5,4}},
{ {2,0}, {2,0.5}, {2,1}, {2,1.5}, {2,2}, {2,2.5}, {2,3}, {2,3.5}, {2,4}},
{ {2.5,0}, {2.5,0.5}, {2.5,1}, {2.5,1.5}, {2.5,2}, {2.5,2.5}, {2.5,3}, {2.5,3.5},
{2.5,4}},
{ {3,0}, {3,0.5}, {3,1}, {3,1.5}, {3,2}, {3,2.5}, {3,3}, {3,3.5}, {3,4}},
{ {3.5,0}, {3.5,0.5}, {3.5,1}, {3.5,1.5}, {3.5,2}, {3.5,2.5}, {3.5,3}, {3.5,3.5},
{3.5,4}},
{ {4,0}, {4,0.5}, {4,1}, {4,1.5}, {4,2}, {4,2.5}, {4,3}, {4,3.5}, {4,4}},      };

struct coord lookup_2[ROW][COL]={
{ {0,0}, {0,0.5}, {0,1}, {0,1.5}, {0,2}, {0,2.5}, {0,3}, {0,3.5}, {0,4}},
{ {0.5,0}, {0.5,0.5}, {0.5,1}, {0.5,1.5}, {0.5,2}, {0.5,2.5}, {0.5,3}, {0.5,3.5},
{0.5,4}},
{ {1,0}, {1,0.5}, {1,1}, {1,1.5}, {1,2}, {1,2.5}, {1,3}, {1,3.5}, {1,4}},
{ {1.5,0}, {1.5,0.5}, {1.5,1}, {1.5,1.5}, {1.5,2}, {1.5,2.5}, {1.5,3}, {1.5,3.5},
{1.5,4}},
{ {2,0}, {2,0.5}, {2,1}, {2,1.5}, {2,2}, {2,2.5}, {2,3}, {2,3.5}, {2,4}},
{ {2.5,0}, {2.5,0.5}, {2.5,1}, {2.5,1.5}, {2.5,2}, {2.5,2.5}, {2.5,3}, {2.5,3.5},
{2.5,4}},
{ {3,0}, {3,0.5}, {3,1}, {3,1.5}, {3,2}, {3,2.5}, {3,3}, {3,3.5}, {3,4}},
{ {3.5,0}, {3.5,0.5}, {3.5,1}, {3.5,1.5}, {3.5,2}, {3.5,2.5}, {3.5,3}, {3.5,3.5},
{3.5,4}},
{ {4,0}, {4,0.5}, {4,1}, {4,1.5}, {4,2}, {4,2.5}, {4,3}, {4,3.5}, {4,4}},      };

double lookup_F1_X[101]={4, 3.8, 3.5, 3.4, 3.3, 3.2, 3.1, 3, 3, 2.9, 2.9, 2.8, 2.8,
2.7, 2.7, 2.6, 2.6, 2.5, 2.5, 2.5, 2.5, 2.4, 2.4, 2.4, 2.3, 2.3, 2.2, 2.2, 2.1, 2.1, 2.1, 2, 2,
2, 2, 1.9, 1.9, 1.9, 1.9, 1.9, 1.8, 1.8, 1.8, 1.8, 1.8, 1.7, 1.7, 1.7, 1.7, 1.7, 1.6, 1.6, 1.6, 1.6,
1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.4, 1.4, 1.4, 1.4, 1.4, 1.4, 1.3, 1.3, 1.3, 1.3, 1.3, 1.3, 1.2, 1.2,
1.2, 1.2, 1.2, 1.2, 1.2, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1, 1, 1, 1, 1, 1, 1, .9, .8, .7, .6,
.5, .4, .3, .2, .1, 0};
double lookup_F1_Y[101]={4, 3.8, 3.6, 3.5, 3.4, 3.2, 3, 3, 2.9, 2.9, 2.8, 2.8, 2.7,
2.7, 2.6, 2.6, 2.5, 2.5, 2.5, 2.4, 2.4, 2.3, 2.3, 2.3, 2.2, 2.2, 2.2, 2.1, 2.1, 2.1, 2, 2, 2, 2,
1.9, 1.9, 1.9, 1.9, 1.9, 1.8, 1.8, 1.8, 1.8, 1.7, 1.7, 1.7, 1.7, 1.7, 1.6, 1.6, 1.6, 1.6, 1.6,
1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.4, 1.4, 1.4, 1.4, 1.4, 1.4, 1.3, 1.3, 1.3, 1.3, 1.3, 1.3, 1.2,
1.2, 1.2, 1.2, 1.2, 1.2, 1.2, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1, 1, 1, 1, 1, 1, 1, .9,
.8, .7, .6, .5, .4, .2, 0};
double lookup_F2_X[101]={0, .5, 1, 1, 1.2, 1.2, 1.4, 1.4, 1.5, 1.5, 1.5, 1.6, 1.6, 1.6,
1.8, 1.8, 1.8, 1.9, 1.9, 1.9, 2, 2, 2, 2, 2.2, 2.2, 2.2, 2.2, 2.2, 2.3, 2.3, 2.3, 2.3, 2.3, 2.3,
2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.7, 2.7, 2.7, 2.7, 2.7, 2.7, 2.7, 2.7, 2.8, 2.8, 2.8, 2.8, 2.8, 2.8,
2.8, 2.8, 2.8, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3.2, 3.2, 3.2, 3.2, 3.2, 3.2, 3.2, 3.3, 3.3, 3.3, 3.3, 3.3,
3.3, 3.3, 3.3, 3.3, 3.3, 3.3, 3.4, 3.4, 3.4, 3.4, 3.4, 3.4, 3.4, 3.4, 3.4, 3.4, 3.5, 3.5,
3.5, 3.5, 3.8, 3.8, 3.8, 4};
double lookup_F2_Y[101]={0, .5, .5, .8, .8, 1, 1, 1, 1.2, 1.2, 1.2, 1.3, 1.3, 1.3, 1.3,
1.5, 1.5, 1.5, 1.7, 1.7, 1.7, 1.7, 1.8, 1.8, 1.8, 1.8, 1.8, 1.8, 2, 2, 2, 2, 2, 2, 2.1, 2.1, 2.1, 2.1, 2.1,
2.2, 2.2, 2.2, 2.2, 2.2, 2.2, 2.3, 2.3, 2.3, 2.3, 2.3, 2.5, 2.5, 2.5, 2.5, 2.5, 2.6, 2.6, 2.6,
2.6, 2.6, 2.7, 2.7, 2.7, 2.7, 2.7, 2.7, 2.8, 2.8, 2.8, 2.8, 2.8, 2.8, 2.8, 2.8, 2.9, 2.9, 2.9,
2.9, 2.9, 2.9, 2.9, 3, 3, 3, 3, 3, 3, 3, 3, 3.2, 3.2, 3.2, 3.2, 3.2, 3.2, 3.4, 3.4, 3.4,
3.4, 3.4, 3.5, 3.8, 4};

double lookup_F1_X2[101]={4, 3.8, 3.7, 3.5, 3.4, 3.2, 3.2, 3, 3, 3, 2.8, 2.8, 2.8, 2.8,
2.6, 2.6, 2.6, 2.5, 2.5, 2.5, 2.5, 2.3, 2.3, 2.3, 2.3, 2.2, 2.2, 2.2, 2.2, 2, 2, 2, 2, 2, 2, 2,
1.8, 1.8, 1.8, 1.8, 1.8, 1.8, 1.8, 1.8, 1.7, 1.7, 1.7, 1.7, 1.7, 1.7, 1.7, 1.5, 1.5, 1.5, 1.5, 1.5,
1.5, 1.3, 1.3, 1.3, 1.3, 1.3, 1.3, 1.3, 1.3, 1.3, 1.3, 1.2, 1.2, 1.2, 1.2, 1.2, 1.2, 1.2, 1.2,
1.2, 1.2, 1.2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0.8, 0.8, 0.8, 0.8, 0.7, 0.7, 0.7, 0.7, 0.5, 0.5,
0.5, 0.2, 0.2, 0};
double lookup_F1_Y2[101]={4, 3.8, 3.7, 3.5, 3.4, 3.2, 3, 3, 2.8, 2.8, 2.8, 2.7, 2.7,
2.7, 2.5, 2.5, 2.5, 2.5, 2.3, 2.3, 2.3, 2.3, 2.3, 2.2, 2.2, 2.2, 2.2, 2, 2, 2, 2, 2, 1.8, 1.8,
1.8, 1.8, 1.8, 1.8, 1.8, 1.8, 1.7, 1.7, 1.7, 1.7, 1.7, 1.7, 1.7, 1.7, 1.5, 1.5, 1.5,
1.5, 1.5, 1.5, 1.5, 1.3, 1.3, 1.3, 1.3, 1.3, 1.3, 1.3, 1.3, 1.3, 1.2, 1.2, 1.2, 1.2, 1.2,
1.2, 1.2, 1.2, 1.2, 1.2, 1.2, 1.2, 1.2, 1.2, 1.2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0.8, 0.8,
0.8, 0.5, 0.5, 0.5, 0.2, 0};

struct coord lookup_F1_X3[11][11]={
{ {3,3}, {3,2.5}, {3,2.25}, {3,2}, {3,1.75}, {3,1.5}, {3,1.25}, {3,1.25}, {3,1},
{3,0.5}, {3,0}},
{ {2.5,3}, {2.5,2.5}, {2.5,2.25}, {2.5,2}, {2.5,1.75}, {2.5,1.5}, {2.5,1.25},
{2.5,1.25}, {2.5,1}, {2.5,0.5}, {2.5,0}},
```

```
{ {2.25,3}, {2.5,2.5}, {2.5,2.25}, {2.5,2}, {2.5,1.75}, {2.5,1.5}, {2.5,1.25},
{2.5,1.25}, {2.5,1}, {2.5,0.5}, {2.5,0}},
{ {2,3}, {2,2.5}, {2,2.25}, {2,2}, {2,1.75}, {2,1.5}, {2,1.25}, {2,1.25}, {2,1},
{2,0.5}, {2,0}},
{ {1.75,3}, {1.75,2.5}, {1.75,2.25}, {1.75,2}, {1.75,1.75}, {1.75,1.5},
{1.75,1.25}, {1.75,1.25}, {1.75,1}, {1.75,0.5}, {1.75,0}},
{ {1.5,3}, {1.5,2.5}, {1.5,2.25}, {1.5,2}, {1.5,1.75}, {1.5,1.5}, {1.5,1.25},
{1.5,1.25}, {1.5,1}, {1.5,0.5}, {1.5,0}},
{ {1.25,3}, {1.25,2.5}, {1.25,2.25}, {1.25,2}, {1.25,1.75}, {1.25,1.5},
{1.25,1.25}, {1.25,1.25}, {1.25,1}, {1.25,0.5}, {1.25,0}},
{ {1.25,3}, {1.25,2.5}, {1.25,2.25}, {1.25,2}, {1.25,1.75}, {1.25,1.5},
{1.25,1.25}, {1.25,1.25}, {1.25,1}, {1.25,0.5}, {1.25,0}},
{ {1,3}, {1,2.5}, {1,2.25}, {1,2}, {1,1.75}, {1,1.5}, {1,1.25}, {1,1.25}, {1,1},
{1,0.5}, {1,0}},
{ {0.5,3}, {0.5,2.5}, {0.5,2.25}, {0.5,2}, {0.5,1.75}, {0.5,1.5}, {0.5,1.25},
{0.5,1.25}, {0.5,1}, {0.5,0.5}, {0.5,0}},
{ {0,3}, {0,2.5}, {0,2.25}, {0,2}, {0,1.75}, {0,1.5}, {0,1.25}, {0,1.25}, {0,1},
{0,0.5}, {0,0}}
    };

/* From Vel2.c */
double forward_multiplier = MAG_MULTI, turning_multiplier = DIR_MULTI;
double magnitude = 0, direction = (PI / 2);
double delta_X, delta_Y;
double delta_X1, delta_X2, delta_Y1, delta_Y2;

double turning_offset = 0;
//double magnitude_x = 0, magnitude_y = 0;

int type = 1;

int DebugRunNumber=0;
double DebugMotion=0;

double TempVoltage[6][8][4] = {
{ {2,6,6,2}, {2,5,6,3}, {2,4,6,4}, {2,3,6,5}, {2,2,6,6}, {2,3,6,5}, {2,4,6,4},
{2,5,6,3}},
{ {2,6,6,6}, {2,5,6,5}, {2,4,6,4}, {2,3,6,3}, {2,2,6,2}, {2,2,6,2}, {2,2,6,2},
{2,2,6,2}},
{ {2,2,6,2}, {2,3,6,3}, {2,4,6,4}, {2,5,6,5}, {2,6,6,6}, {2,5,6,5}, {2,4,6,4},
{2,3,6,3}},
{ {2,4,6,2}, {2,4,6,3}, {2,4,6,4}, {2,4,6,5}, {2,4,6,6}, {2,4,6,5}, {2,4,6,4},
{2,4,6,3}},
{ {2,2,6,4}, {2,3,6,4}, {2,4,6,4}, {2,5,6,4}, {2,6,6,4}, {2,5,6,4}, {2,4,6,4},
{2,3,6,4}},
{ {0,6,8,2}, {1,5,7,3}, {2,4,6,4}, {3,3,5,5}, {4,2,4,6}, {3,3,5,5}, {2,4,6,4},
{1,5,7,3}} };




/*-------------*/
/* From a2d.c */
/*-------------*/

int get_finger(void /* int run_number*/)
{
    initialize_A2D();

    if(run_number==0)
    {
        //while(error==TRUE)
        error = get_values();

        //while(error==TRUE)
        error = lookup_coordinates();

        store_last();
        run_number++;
    }

//  while(error==TRUE)
        error = get_values();

//  while(error==TRUE)
        error = lookup_coordinates();

//  while(error==TRUE)
        error = get_mag_and_dir();

//  while(error==TRUE)
        error = store_last();

    close_driver();

    return error;
}

int get_finger_debug(void)
{
    int ab;
/*
    if(run_number==0)
    {
        for (ab=0; ab < 4; ab++)
        {
        data[ab] = TempVoltage[DebugMotion][DebugRunNumber % 8][ab];
        }
    }
    DebugRunNumber++;

    while(error==TRUE)
        error = lookup_coordinates();
*/

    for (ab=0; ab < 4; ab++)
    {
        data[ab] = TempVoltage[((int)DebugMotion)][DebugRunNumber %
8][ab];
    }

    DebugRunNumber++;

//  while(error==TRUE)
        error = lookup_coordinates2();

//  while(error==TRUE)
        error = get_mag_and_dir();

//          TRACE("new:,  (%.3f, %.3f)  (%.3f, %.3f) ", first.x, first.y,
old_first.x, old_first.y);
//  while(error==TRUE)
        error = store_last();

    return error;
}

/*
int a2d_coordinates(void)
{
    int a = 0;
    // Initialize the hardware and software
    if(( Err = DAS1600_DevOpen( "board1.CFG", &NumOfBoards )) != 0)
    {
        putch (7);
//      printf("Error %X during DevOpen", Err);
        exit(Err);
    }

    // Establish communication with the driver through a device handle
    if(( Err = DAS1600_GetDevHandle( 0, &DAS1600)) != 0)
    {
        putch (7);
//      printf("Error %X during GetDevHandle ", Err);
        exit(Err);
    }

    for (a=0; a < 4; a++)
    {

        // Read channel 0 at gain 1; store sample in Advalue
        if((Err = K_ADRead(DAS1600, a, 0, &Advalue)) != 0)
        {
            putch (7);
//          printf("Error in K_ADRead operation.");
            exit(Err);
        }

        // Strip channel tag and display ADvalue
        // printf("A/D value from channel 0 is: %x\n", ADvalue>>4);
```

```
        data[a] = Advalue>>4;
        data[a] = (((data[a]) - 2048) * (20)) / (4096);

    }
    return 0;
}
*/


int lookup_coordinates(void)
{
    int a = 0;
    int temp[4];

    for (a=0; a < 4; a++)
    {
        if(data[a]<0)
            data[a]=0;
        if(data[a]>10)
            data[a]=10;

        temp[a] = (int)(10*data[a]);
    }

// For a 2by2 lookup table

//  first = lookup_F1_X3[temp[0]][temp[1]];
//  second = lookup_F1_X3[temp[2]][temp[3]];

// For a 1by1 lookup table
    coords[0] = first.x=  lookup_F1_X2[temp[0]];
    coords[1] = first.y=  lookup_F1_Y2[temp[1]];
    coords[2] = second.x= lookup_F2_X[temp[2]];
    coords[3] = second.y= lookup_F2_Y[temp[3]];

    TRACE("V2C  (%.3f - %d - %.2f) (%.3f - %d - %.2f) (%.3f - %d - %.2f)
(%.3f - %d - %.2f) \n", data[0], temp[0], first.x, data[1], temp[1], first.y, data[2],
temp[2], second.x, data[3], temp[3], second.y);



    return 0;
}
int lookup_coordinates2(void)
{
    int a = 0;
    int temp[4];

    for (a=0; a < 4; a++)
    {
        temp[a] = (int)data[a];
    }

// For a 2by2 lookup table

    first = lookup_1[temp[0]][temp[1]];
    second = lookup_2[temp[2]][temp[3]];

// For a 1by1 lookup table
    coords[0] = first.x;//=  lookup_F1_X[temp[0]];
    coords[1] = first.y;//=  lookup_F1_Y[temp[1]];
    coords[2] = second.x;//= lookup_F2_X[temp[2]];
    coords[3] = second.y;//= lookup_F2_Y[temp[3]];

    return 0;
}
/*------------------*/
/* From Storpast.c  */
/*------------------*/


/* function stores the new values into the old values */

int store_last(void)
{

    old_first = first;
    old_second = second;

/*  printf("\n new:  (%i, %i)  (%i, %i)", first.x, first.y, second.x, second.y);
    printf("\n old:  (%i, %i)  (%i, %i)", old_first.x, old_first.y, old_second.x,
old_second.y);
*/
    return 0;
}

/*------------------*/
/* From stormult.c */
/*------------------*/


/* function stores the new values into the array of old values */

int store_multiple(void)
{
    int a = 0, b=0;

    old_first = first;
    old_second = second;

    for(b=(SAVE-1); b>1 ; b--)
    {
        for(a=0; a<4 ; a++)
        {
            old_coords[b][a] = old_coords[b-1][a];
        }
    }
    for(a=0; a<4 ; a++)
    {
        old_coords[0][a] = coords[a];
    }

/*  printf("\n coords:%.4f, %.4f, %.4f, %.4f", coords[0], coords[1], coords[2],
coords[3]);
    for(a=0; a<=4; a++)
    {
        printf("\n 0: %.4f, %.4f, %.4f, %.4f", old_coords[c][0],
old_coords[c][1], old_coords[c][2], old_coords[c][3]);
    }
*/
    return 0;
}


/*-------------*/
/* From Vel3.c */
/*-------------*/


/* function determines the instantaneous magnitude of the
   velocity vector and returns it as a double.  also sets
   global variable magnitude to the correct value */

/* Type             0 = normal
                    1 = nothing
                    2 = backwards
                    3 = right turn
                    4 = left turn

   Finger 1 = fore finger (right hand), middle finger (left hand)
   Finger 2 = middle finger (right hand), fore finger (left hand)
*/

int get_mag_and_dir(void)
{
    double temp_X=0, temp_Y=0;
    double temp_X1=0, temp_X2=0;
    double temp_Y1=0, temp_Y2=0;
    double m = 0;

    temp_X1 = get_delta_X1(); // determine x component of Finger 1 velocity
    temp_X2 = get_delta_X2(); // determine x component of Finger 2 velocity
    temp_Y1 = get_delta_Y1(); // determine y component of Finger 1 velocity
    temp_Y2 = get_delta_Y2(); // determine y component of Finger 2 velocity

    if( temp_Y1 > 0) // Finger 1 velocity in forward direction
    {
        if( temp_Y2 > 0)            // Both Finger 1 and Finger 2 in
Forward direction
            type = 1;               // mode of operation:
Nothing
```

```
        else if( temp_Y2 < 0)// Finger 1 forward, finger 2 backwards
                type = 0;                       // mode of operation:
Normal
        else                    // if temp_Y2 = 0:  Finger 1 forward, Finger 2
stationary
                type = 4;                       // mode of operation:  left
turn
    }
    else if( temp_Y1 < 0)       //Finger 1 velocity in backwards direction
    {
        if( temp_Y2 > 0)                // Finger 1 backwards, Finger 2
forwards
                type = 0;                       // mode of operation:
Normal
        else if( temp_Y2 < 0)// Both finger 1 and Finger 2 backwards
                type = 2;                       // mode of operation:
backwards
        else                    // if temp_Y2 = 0:  Finger 1 backwards, Finger 2
stationary
                type = 4;                       // mode of operation:  left
turn
    }
    else                        // if temp_Y1 = 0:  Finger 1 stationary
    {
        if( temp_Y2 > 0)                // Finger 1 stationary, Finger 2
Forward
                type = 3;                       // mode of operation:  Right
Turn
        else if( temp_Y2 < 0)// Finger 1 stationary, Finger 2 backwards
                type = 3;                       // mode of operation:  Right
Turn
        else                    // if temp_Y2 = 0:  Both Finger 1 and Finger 2
stationary
                type = 1;                       // mode of operation:
nothing
    }

    temp_X = get_velocity_delta_X(type); // get velocity in x direction
    temp_Y = get_velocity_delta_Y(type); // get velocity in y direction

    switch (type)
    {
    case 0:             // Normal
        {
                magnitude =             ((temp_X) * (temp_X)) + ((temp_Y)
* (temp_Y));
                magnitude = sqrt(magnitude);
                magnitude = magnitude * forward_multiplier;

                if( temp_X != 0)
                {
                        if( temp_Y != 0)                        //
atan2 returns an error if point at origin
                                direction = atan2(temp_Y, temp_X);
        // range of atan2 -pi to pi
                        else
                                direction = (-1) * PI/2;
                }
                else
                        direction = (-1) * PI/2;

//              TRACE("DX %.2lf DY %.2lf DIR %.4lf \n", temp_X,
temp_Y, direction);

                direction = direction - PI;                     //
actual walking in opposite direction of movement
                direction = fmod(direction + turning_offset, (2*PI));

//              TRACE("adjDIR:  %.4lf  Off:  %.4lf \n", direction,
turning_offset);

                break;
        }
    case 1:             // Nothing
        {
                magnitude = 0;
                direction = direction; // maintain old direction

                break;
        }
    case 2:             // Backwards
        {
                magnitude =             ((temp_X) * (temp_X)) + ((temp_Y)
* (temp_Y));
                magnitude = sqrt(magnitude);
                magnitude = magnitude * forward_multiplier;
                magnitude = (-1.0) * magnitude;

//              TRACE("TEMPS: (%.4lf, %.4lf) Mag:  %.4lf", temp_X,
temp_Y, magnitude);

                if( temp_X != 0)
                {
                        if( temp_Y != 0)                        //
atan2 returns an error if point at origin
                                direction = atan2(temp_Y, temp_X);
        // range of atan2 -pi to pi
                        else
                                direction = (-1) * PI/2;
                }
                else
                        direction = (-1) * PI/2;

                direction = direction - PI;
                direction = fmod(direction + turning_offset, (2*PI));

//              TRACE("adjDIR:  %.4lf  Off:  %.4lf \n", direction,
turning_offset);


                break;
        }
    case 3:             // right turn
        {
                magnitude = 0;

                if(temp_Y < 0)
                {
                        m =             ((temp_X) * (temp_X)) + ((temp_Y)
* (temp_Y));
                        m = sqrt(m);
                }
                else
                        m = 0;
                direction = ((m * turning_multiplier) * PI);
                turning_offset = fmod(turning_offset - direction, (2*PI));
                direction = fmod((PI / 2) + turning_offset, (2*PI));

                break;
        }
    case 4:             // left turn
        {
                magnitude = 0;

                if(temp_Y < 0)
                {
                        m =             ((temp_X) * (temp_X)) + ((temp_Y)
* (temp_Y));
                        m = sqrt(m);
                }
                else
                        m=0;
                direction = ((m * turning_multiplier) * PI);
                turning_offset = fmod(direction + turning_offset, (2*PI));
                direction = fmod((PI / 2) + turning_offset, (2*PI));

                break;
        }
    default:
        {
                magnitude = 0;                                  // 0 velocity
                direction = direction; // maintain previous direction

        }
    }

    return 0;
}

// determines the movement velocity depending on the type of operation
double get_velocity_delta_X(int type)
{
    switch (type)
```

```
    {
    case 0:                                 // opposite vel--Normal
            {
                    if (delta_Y1 > 0)                       // Finger 1
moving forward
                            delta_X = delta_X2;             // use finger 2
for velocity
                    else
    // Finger 2 moving forward
                            delta_X = delta_X1;             // use finger 1
for velocity
                    break;
            }
    case 1:                                 // both forward--Nothing
            {
                    delta_X = 0;                                    //
velocity = 0
                    break;
            }
    case 2:                                 // both backwards--Backwards
            {
                    delta_X = ((delta_X1 + delta_X2) / 2);  // average
movement of both fingers
                    break;
            }
    case 3:                                 // Y1 = 0--Right Turn
            {
                    delta_X = delta_X2;                     // use finger 2
velocity to find turning factor
                    break;
            }
    case 4:                                 // Y2 = 0--left turn
            {
                    delta_X = delta_X1;                     // use finger 1
velocity to find turning factor
                    break;
            }
    default:
            {
                    delta_X = 0;                                    // x
velocity = 0
            }
    }

    return delta_X;
}


double get_velocity_delta_Y(int type)
{
    switch (type)
    {
    case 0:                                 // opposite vel--Normal
            {
                    if (delta_Y1 < 0)                       // Finger 1
moving backwards
                            delta_Y = delta_Y1;             // Use finger 1
to determine velocity
                    else
                            delta_Y = delta_Y2;             // Use finger 2
to determine velocity
                    break;
            }
    case 1:                                 // both forward--nothing
            {
                    delta_Y = 0;                                    //
velocity = 0
                    break;
            }
    case 2:                                 // both backwards--Backwards
            {
                    delta_Y = ((delta_Y1 + delta_Y2) / 2);  // average
movement of both fingers
                    break;
            }
    case 3:                                 // Y1 = 0--right turn
            {
                    delta_Y = delta_Y2;                     // use finger 2
velocity to find turning factor
                    break;
            }
    case 4:                                 // Y2 = 0--Left turn
            {
                    delta_Y = delta_Y1;                     // use finger 1
velocity to find turning factor
                    break;
            }
    default:
            {
                    delta_Y = 0;                                    // y
velocity = 0
            }
    }

    return delta_Y;
}

double get_delta_X1(void)
{
    delta_X1 = (first.x) - (old_first.x);
    return delta_X1;
}

double get_delta_X2(void)
{
    delta_X2 = (second.x) - (old_second.x);
    return delta_X2;
}

double get_delta_Y1(void)
{
    delta_Y1 = (first.y) - (old_first.y);
    return delta_Y1;
}

double get_delta_Y2(void)
{
    delta_Y2 = (second.y) - (old_second.y);
    return delta_Y2;
}

// From Testbed software
int get_values(void)
{
    int a = 0;


    for (a=0; a < 4; a++)
    {

        // Read channel 0 at gain 1; store sample in Advalue
        if((nErr = K_ADRead(hDev1600, a, 0, &wADval)) != 0)
        {
                putchar(7);
                printf("Error %X in K_ADRead operation.", nErr);
                exit(1);
        }

        // Strip channel tag and display ADvalue
        // printf("A/D value from channel 0 is: %x\n", (wADval>>4)&0xfff);

        Counts[a] = (wADval>>4)&0xfff;
        data[a] = ((((double)Counts[a]) - 2048) * (20)) / (4096);
    }

    return 0;
}

int initialize_A2D(void)
{
    // Initialize the hardware and software
    if(( nErr = K_OpenDriver( "DAS1600", "das1600.CFG", &hDrv1600)) != 0)
    {
        putchar(7);
        printf("Error %X during K_OpenDriver", nErr);
        exit(nErr);
    }

    // Establish communication with the driver through a device handle
    if(( nErr = K_GetDevHandle(hDrv1600, 0, &hDev1600)) != 0)
    {
        putchar(7);
```

```
        printf("Error %X during K_GetDevHandle ", nErr);
        exit(1);
    }
    return 0;
}

int close_driver(void)
{
    // Close the Driver and Release All Resources
    K_CloseDriver(hDrv1600);

    return 0;
}
```

```
/*****************************************************************
******

Program:        ADTracker

File:           finger.h

Function:       Header for all finger walker code

Description:

Author:         Brian Fitch (BF)

Environment:    Microsoft Visual C++ 5.0, 486/66 16mb RAM, Windows 95
                DOS 6.0.

Notes:          For use with finger.cpp

Revisions:      1.00    3/10/98   (BF)    Initial Release
*****************************************************************
******/

#ifndef __FINGER_H__
#define __FINGER_H__

//from a2d card header
#define IDS_ERR_REGISTER_CLASS  1
#define IDS_ERR_CREATE_WINDOW   2
#define STARTBTN            101
#define STOPBTN             102
#define CLOSEBTN            118
#define CONTBOX             124
#define STARTCHANBOX        104
#define STOPCHANBOX         105
#define NUMSAMPLESBOX       108
#define SAMPLERATEBOX       110
#define DMASTATUSBOX        113
#define DMATRANSFERBOX      114
#define DATALISTBOX         119
#define ID_TIMER            1

//char acString[128];  /* variable to load resource strings      */
//char *szString = acString;

//char *szAppName = "              ";  // class name for the window

//HWND hInst;
//HWND hWndMain;

LONG FAR PASCAL WndProc(HWND, UINT, UINT, LONG);
int  nCwRegisterClasses(void);
void CwUnRegisterClasses(void);
void InitWindowFields( HWND hwnd );
void InitDASDevice(void) ;
void StartAcquiring(HWND hWnd) ;
void StopAcquiring(HWND hWnd) ;
void ProcessTimer(HWND hWnd);
void ShowData(HWND hWnd) ;
void ProcessError(short ErrNum) ;

/*---------------------------*/
/* Function Prototypes     */
/*---------------------------*/

/* For Running */
int get_finger(void);
int get_finger_debug(void);

/* From A2D*/
int get_coordinates(void);
//int a2d_coordinates(void);
int lookup_coordinates(void);
int lookup_coordinates2(void);

// From Testbed software
int get_values(void);
int close_driver(void);
int initialize_A2D(void);


/* From Storpast.c */
int store_last(void);
int store_multiple(void);

/*From Vel2.c */
//double get_magnitude(void);
//double get_direction(void);
int get_mag_and_dir(void);
double get_velocity_delta_X(int type);
double get_velocity_delta_Y(int type);
double get_delta_X1(void);
double get_delta_X2(void);
double get_delta_Y1(void);
double get_delta_Y2(void);

/*---------------------------*/
/* #define directives      */
/*---------------------------*/

/* From a2d.c */
#define TRUE        1
#define FALSE       0
#define COL                 101
#define ROW                 101
#define SAVE        10
#define TURNING     1
#define MAG_MULTI           1
#define DIR_MULTI 1
#define PI          3.141592654

/*---------------------------*/
/* Definitions of structures*/
/* and variables          */
/*---------------------------*/

extern DWORD hDrv1600;                  // Driver Handle
extern DWORD hDev1600;                  // Device Handle
extern short nErr;                      // Function return error
flag
extern WORD wADval;                     // Storage for A/D value

extern int error;                   // error = 0 :: no errors
                                        // error= 1 :: system error


extern int Counts[4];
extern double data[4];                  /*  data[0] == x1

    data[1] == y1

    data[2] == x2

    data[3] == y2
                                                              */
extern double coords[4];

extern double old_coords[SAVE][4];

struct coord {
    double x;
    double y;
};

extern struct coord first;
extern struct coord second;
extern struct coord old_first;
extern struct coord old_second;
```

```
// lookup[X][Y]
extern struct coord lookup_1[ROW][COL];
extern struct coord lookup_2[ROW][COL];
extern struct coord lookup_F1_X3[11][11];

extern double lookup_F1_X[101];
extern double lookup_F1_Y[101];
extern double lookup_F2_X[101];
extern double lookup_F2_Y[101];

extern double lookup_F1_X2[101];
extern double lookup_F1_Y2[101];

/* From Vel2.c */
extern double forward_multiplier;
extern double turning_multiplier;
extern double magnitude;
extern double direction;
extern double delta_X;
extern double delta_Y;
extern double delta_X1;
extern double delta_X2;
extern double delta_Y1;
extern double delta_Y2;

extern double turning_offset;
//double magnitude_x = 0, magnitude_y = 0;

extern int type;

extern int DebugRunNumber;
extern double DebugMotion;

extern double TempVoltage[6][8][4];


#endif // __FINGER_H__
```

H Demo Code

The following are the C++ listings for the ADTracker: ADTracker.cpp, ADTracker.h, ADTrackerDlg.cpp, ADTrackerDlg.h, ModifyConstantsDlg.cpp, ModifyConstantsDlg.h, resource.h, StdAfx.cpp, StdDfx.h, TrackerDlg.cpp, and TrackerDlg.h.

```
// ADTracker.cpp : Defines the class behaviors for the application.
//


#include "stdafx.h"
#include "ADTracker.h"
#include "ADTrackerDlg.h"


#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CADTrackerApp

BEGIN_MESSAGE_MAP(CADTrackerApp, CWinApp)
    //{{AFX_MSG_MAP(CADTrackerApp)
        // NOTE - the ClassWizard will add and remove mapping macros here.
        //    DO NOT EDIT what you see in these blocks of generated code!
    //}}AFX_MSG
    ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CADTrackerApp construction

CADTrackerApp::CADTrackerApp()
{
    // TODO: add construction code here,
    // Place all significant initialization in InitInstance
}

/////////////////////////////////////////////////////////////////////////////
// The one and only CADTrackerApp object

CADTrackerApp theApp;

/////////////////////////////////////////////////////////////////////////////
// CADTrackerApp initialization

BOOL CADTrackerApp::InitInstance()
{
    // Standard initialization
    // If you are not using these features and wish to reduce the size
    //  of your final executable, you should remove from the following
    //  the specific initialization routines you do not need.

#ifdef _AFXDLL
    Enable3dControls();                     // Call this when using MFC
in a shared DLL
#else
    Enable3dControlsStatic();   // Call this when linking to MFC statically
#endif

    dlg = new CADTrackerDlg();
    m_pMainWnd = dlg;
    int nResponse = dlg->DoModal();
    if (nResponse == IDOK)
    {
        // TODO: Place code here to handle when the dialog is
        //  dismissed with OK
    }
    else if (nResponse == IDCANCEL)
    {
        // TODO: Place code here to handle when the dialog is
        //  dismissed with Cancel
    }

    delete dlg;
    // Since the dialog has been closed, return FALSE so that we exit the
    //  application, rather than start the application's message pump.
    return FALSE;
}

CADTrackerDlg* CADTrackerApp::GetDlg()
{
    return theApp.dlg;
}

// ADTracker.h : main header file for the ADTracker application
//


#ifndef __AFXWIN_H__
    #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"       // main symbols
#include "ADTrackerDlg.h"

/////////////////////////////////////////////////////////////////////////////
// CADTrackerApp:
// See ADTracker.cpp for the implementation of this class
//

class CADTrackerApp : public CWinApp
{
public:
    CADTrackerApp();

    static CADTrackerDlg* GetDlg();


    CADTrackerDlg *dlg;
// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CADTrackerApp)
    public:
    virtual BOOL InitInstance();
    //}}AFX_VIRTUAL

// Implementation

    //{{AFX_MSG(CADTrackerApp)
```

```
        // NOTE - the ClassWizard will add and remove member functions
here.
        //    DO NOT EDIT what you see in these blocks of generated code !
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

// ADTrackerDlg.cpp : implementation
//


#include "stdafx.h"

#include "ADTracker.h"

#include "ADTrackerDlg.h"

#include <math.h>

#include "finger.h"


#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

UINT MultiThreadRun( LPVOID pParam );
UINT MultiThreadDebug( LPVOID pParam );



/////////////////////////////////////////////////////////////////////////////
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
    CAboutDlg();

// Dialog Data
    //{{AFX_DATA(CAboutDlg)
    enum { IDD = IDD_ABOUTBOX };
    //}}AFX_DATA

    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CAboutDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV
support
    //}}AFX_VIRTUAL

// Implementation
protected:
    //{{AFX_MSG(CAboutDlg)
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
    //{{AFX_DATA_INIT(CAboutDlg)
    //}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CAboutDlg)
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
    //{{AFX_MSG_MAP(CAboutDlg)
        // No message handlers
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CADTrackerDlg dialog

CADTrackerDlg::CADTrackerDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CADTrackerDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CADTrackerDlg)
        // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
    // Note that LoadIcon does not require a subsequent DestroyIcon in Win32
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
    CONTINUE = TRUE;
    OriginX = 30;
    OriginY = 235;
    length = 320;
    MaxVal = 4.0;
    CompassOriginX = 260;
    CompassOriginY = 100;
    CompassLength = 100;
    CompassCenterX = CompassOriginX + CompassLength/2;
    CompassCenterY = CompassOriginY + CompassLength/2;

    sprintf(ErrorMessages[0], "Normal");
    sprintf(ErrorMessages[1], "Nothing");
    sprintf(ErrorMessages[2], "Reverse");
    sprintf(ErrorMessages[3], "Right");
    sprintf(ErrorMessages[4], "Left");
    sprintf(ErrorMessages[5], "Error");
    type = 1;

    ConstantsDlg = new ModifyConstantsDlg(this);
    ConstantsDlg->Create( IDD_DIALOG_CONSTANTS, this );

    TrackDlg = new TrackerDlg(this);
    TrackDlg->Create( IDD_DIALOG_TRACKER, this );
    TrackDlg->ShowWindow(SW_SHOW);

    TrackerScale = 1;
    TrackerCenterX = 250;
    TrackerCenterY = 250;
    TrackerLength = 250;
    NumPoints = 1;
    StartNumPoint = 0;
    ThePoints[0].x = TrackerCenterX;
    ThePoints[0].y = TrackerCenterY;
}

CADTrackerDlg::~CADTrackerDlg()
{
    delete ConstantsDlg;
    delete TrackDlg;
}

void CADTrackerDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CADTrackerDlg)
        // NOTE: the ClassWizard will add DDX and DDV calls here
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CADTrackerDlg, CDialog)
    //{{AFX_MSG_MAP(CADTrackerDlg)
    ON_WM_SYSCOMMAND()
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()
    ON_BN_CLICKED(IDC_BUTTON_RUN, OnButtonRun)
    ON_BN_CLICKED(IDC_BUTTON_STOP, OnButtonStop)
    ON_BN_CLICKED(IDC_BUTTON_QUIT, OnButtonQuit)
    ON_BN_CLICKED(IDC_BUTTON_PARAMETERS, OnButtonParameters)
    ON_BN_CLICKED(IDC_BUTTON_DIALOG, OnButtonDialog)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CADTrackerDlg message handlers
```

```
BOOL CADTrackerDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // Add "About..." menu item to system menu.

    // IDM_ABOUTBOX must be in the system command range.
    ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
    ASSERT(IDM_ABOUTBOX < 0xF000);

    CMenu* pSysMenu = GetSystemMenu(FALSE);
    CString strAboutMenu;
    strAboutMenu.LoadString(IDS_ABOUTBOX);
    if (!strAboutMenu.IsEmpty())
    {
        pSysMenu->AppendMenu(MF_SEPARATOR);
        pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX,
strAboutMenu);
    }

    // Set the icon for this dialog.  The framework does this automatically
    //  when the application's main window is not a dialog
    SetIcon(m_hIcon, TRUE);            // Set big icon
    SetIcon(m_hIcon, FALSE);        // Set small icon

    // TODO: Add extra initialization here

    return TRUE;  // return TRUE  unless you set the focus to a control
}

void CADTrackerDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
    if ((nID & 0xFFF0) == IDM_ABOUTBOX)
    {
        CAboutDlg dlgAbout;
        dlgAbout.DoModal();
    }
    else
    {
        CDialog::OnSysCommand(nID, lParam);
    }
}

// If you add a minimize button to your dialog, you will need the code below
//  to draw the icon.  For MFC applications using the document/view model,
//  this is automatically done for you by the framework.

void CADTrackerDlg::OnPaint()
{
    if (IsIconic())
    {
        CPaintDC dc(this); // device context for painting

        SendMessage(WM_ICONERASEBKGND, (WPARAM)
dc.GetSafeHdc(), 0);

        // Center icon in client rectangle
        int cxIcon = GetSystemMetrics(SM_CXICON);
        int cyIcon = GetSystemMetrics(SM_CYICON);
        CRect rect;
        GetClientRect(&rect);
        int x = (rect.Width() - cxIcon + 1) / 2;
        int y = (rect.Height() - cyIcon + 1) / 2;

        // Draw the icon
        dc.DrawIcon(x, y, m_hIcon);
    }
    else
    {
        CPaintDC dc(this); // device context for painting

        int x1 = (int)(first.x*(length/MaxVal));
        int y1 = (int)(length - first.y*(length/MaxVal));
        int x2 = (int)(second.x*(length/MaxVal));
        int y2 = (int)(length - second.y*(length/MaxVal));

        CPoint Point1(OriginX + x1, OriginY + y1);
        CPoint Point2(OriginX + x2, OriginY + y2);
        dc.FrameRect(&CRect(OriginX, OriginY, OriginX+length,
OriginY+length),
                                    &CBrush(RGB(0,0,0)));

        //Draw Grid for twin points
        dc.FrameRect(&CRect((int)(OriginX),
                                                    (int)(OriginY +
length/MaxVal),
                                                    (int)(OriginX +
length),
                                                    (int)(OriginY +
2*length/MaxVal)),
                                    &CBrush(RGB(0,0,0)));
        dc.FrameRect(&CRect((int)(OriginX),
                                                    (int)(OriginY +
2*length/MaxVal),
                                                    (int)(OriginX +
length),
                                                    (int)(OriginY +
3*length/MaxVal)),
                                    &CBrush(RGB(0,0,0)));
        dc.FrameRect(&CRect((int)(OriginX + length/MaxVal),
                                                    (int)(OriginY),
                                                    (int)(OriginX +
2*length/MaxVal),
                                                    (int)(OriginY +
length)),
                                    &CBrush(RGB(0,0,0)));
        dc.FrameRect(&CRect((int)(OriginX + 2*length/MaxVal),
                                                    (int)(OriginY),
                                                    (int)(OriginX +
3*length/MaxVal),
                                                    (int)(OriginY +
length)),
                                    &CBrush(RGB(0,0,0)));


        //Draw Compass
        dc.FrameRect(&CRect((int)(CompassOriginX),

    (int)(CompassOriginY),

    (int)(CompassOriginX + CompassLength),

    (int)(CompassOriginY + CompassLength)),
                                    &CBrush(RGB(0,0,0)));

        //Draw Axes
        dc.MoveTo(CompassCenterX - CompassLength/4, CompassCenterY);
        dc.LineTo(CompassCenterX + CompassLength/4, CompassCenterY);
        dc.MoveTo(CompassCenterX, CompassCenterY - CompassLength/4);
        dc.LineTo(CompassCenterX, CompassCenterY + CompassLength/4);
        dc.MoveTo(CompassCenterX, CompassCenterY);

        double PointX=0.0, PointY=0.0;

        PointX = (0.4)*CompassLength*cos(direction);
        PointY = (-0.4)*CompassLength*sin(direction);

        //TRACE("dir:  %.4lf  Point:  (%.4lf, %.4lf) \n", direction, PointX,
PointY);
        //TRACE("center:  (%d, %d)   Point+:  (%d, %d) \n", CompassCenterX.
CompassCenterY, (((int)PointX)+CompassCenterX),
(((int)PointY)+CompassCenterY));


        dc.LineTo( (int)(PointX + CompassCenterX), (int)(PointY +
CompassCenterY));

        dc.FillRect(&CRect( (int)(PointX + CompassCenterX), (int)(PointY +
CompassCenterY), (int)(PointX + CompassCenterX+4), (int)(PointY +
CompassCenterY+4)),
                                    &CBrush(RGB(255,0,0)));


        dc.FillRect(&CRect(Point1, CPoint(Point1 + CPoint(4,4))),
                                    &CBrush(RGB(255,0,0)));
        dc.FillRect(&CRect(Point2, CPoint(Point2 + CPoint(4,4))),
                                    &CBrush(RGB(255,0,0)));

        CDialog::OnPaint();
    }
}
```

```
// The system calls this to obtain the cursor to display while the user drags
// the minimized window.
HCURSOR CADTrackerDlg::OnQueryDragIcon()
{
    return (HCURSOR) m_hIcon;
}

void CADTrackerDlg::OnButtonRun()
{
    // TODO: Add your control notification handler code here
    CONTINUE = TRUE;
    AfxBeginThread(MultiThreadRun, NULL);
}

UINT MultiThreadRun( LPVOID pParam )
{
  /*CMyObject* pObject = (CMyObject*)pParam;

  if (pObject == NULL ||
     !pObject->IsKindOf(RUNTIME_CLASS(CMyObject)))
  return -1;            // illegal parameter
*/

    CADTrackerDlg* TheDlg = CADTrackerApp::GetDlg();

    delta_X=2.123456789;
    delta_Y=1.123456789;
    direction=3.141592654;
    magnitude=4.0987654321;
    static int FIRSTTIME = 1;

    CStatic *X1 = (CStatic*)TheDlg->GetDlgItem(IDC_STATIC_X1);
    CStatic *X2 = (CStatic*)TheDlg->GetDlgItem(IDC_STATIC_X2);
    CStatic *Y1 = (CStatic*)TheDlg->GetDlgItem(IDC_STATIC_Y1);
    CStatic *Y2 = (CStatic*)TheDlg->GetDlgItem(IDC_STATIC_Y2);
    CStatic *Mag = (CStatic*)TheDlg->GetDlgItem(IDC_STATIC_MAG);
    CStatic *Dir = (CStatic*)TheDlg->GetDlgItem(IDC_STATIC_DIR);
    CStatic *DeltaX = (CStatic*)TheDlg-
>GetDlgItem(IDC_STATIC_DELTAX);
    CStatic *DeltaY = (CStatic*)TheDlg-
>GetDlgItem(IDC_STATIC_DELTAY);
    CStatic *ErrorNumber = (CStatic*)TheDlg-
>GetDlgItem(IDC_STATIC_NUMBER);
    CStatic *ErrorType = (CStatic*)TheDlg->GetDlgItem(IDC_STATIC_TYPE);


    char X1_text[32];
    char X2_text[32];
    char Y1_text[32];
    char Y2_text[32];
    char Mag_text[32];
    char Dir_text[32];
    char DeltaX_text[32];
    char DeltaY_text[32];
    char StrErrNum[8];

    int count=0;

    while(TheDlg->CONTINUE)
    {
        if(FIRSTTIME)
        {
            FIRSTTIME = 0;
            first.x = (double)3.0987654321;
            first.y = (double)4.0987654321;
            second.x = (double)5.0987654321;
            second.y = (double)6.0987654321;
        }

        double MAX_RAND = 32565.0;

        (double)rand()/MAX_RAND*4.0;
        delta_X = (double)(rand()/MAX_RAND*4.0);
        delta_Y = (double)(rand()/MAX_RAND*4.0);
        direction = (double)(rand()/MAX_RAND*4.0);
        magnitude = (double)(10.0*rand()/MAX_RAND*4.0);
        first.x = (double)(rand()/MAX_RAND*4.0);
        first.y = (double)(rand()/MAX_RAND*4.0);
        second.x = (double)(rand()/MAX_RAND*4.0);
        second.y = (double)(rand()/MAX_RAND*4.0);

        //Tracker Dialog Drawing Stuff
        TheDlg->magnitude_x = magnitude * cos(direction);
        TheDlg->magnitude_y = magnitude * sin(direction);

//      TRACE("magnitude_x,y = (%lf,%lf)\n", TheDlg->magnitude_x,
TheDlg->magnitude_y);

        int NewPt = TheDlg->NumPoints % 2048;
        int PrevPt = TheDlg->NumPoints % 2048;
        if(NewPt == 0)
            PrevPt = 2047;
        else
            PrevPt = NewPt - 1;

        TheDlg->ThePoints[NewPt].x = (int)(TheDlg->ThePoints[PrevPt].x +
TheDlg->magnitude_x);
        TheDlg->ThePoints[NewPt].y = (int)(TheDlg->ThePoints[PrevPt].y +
TheDlg->magnitude_y);
        TRACE("NewPt==%d, (%d,%d)\n", NewPt, TheDlg-
>ThePoints[NewPt].x, TheDlg->ThePoints[NewPt].y);

        TheDlg->NumPoints++;
        if(TheDlg->NumPoints <= 2048)
            TheDlg->StartNumPoint = 0;
        else
            TheDlg->StartNumPoint = (NewPt+1)%2048;

        while(abs(TheDlg->ThePoints[(TheDlg->NumPoints-
1)%2048].x/TheDlg->TrackerScale) > TheDlg->TrackerCenterX + TheDlg-
>TrackerLength ||
              abs(TheDlg->ThePoints[(TheDlg->NumPoints-
1)%2048].y/TheDlg->TrackerScale) > TheDlg->TrackerCenterY + TheDlg-
>TrackerLength)
        {
            TheDlg->TrackerScale *= 2;
        }

        TRACE("TrackerScale==%d\n", TheDlg->TrackerScale);
        if(FALSE)//(get_finger())
        {
            TheDlg->MessageBox("I Failed on Run!\nExiting...");
            //OnOK();
        }
        else //Set all the Text Areas based on the new values
        {
            TRACE("Run Number %d\n", count++);

            sprintf(StrErrNum, "%d", type);
            sprintf(X1_text, "%.3lf", first.x);
            sprintf(X2_text, "%.3lf", second.x);
            sprintf(Y1_text, "%.3lf", first.y);
            sprintf(Y2_text, "%.3lf", second.y);
            sprintf(Mag_text, "%.4lf", magnitude);
            sprintf(Dir_text, "%.4lf", direction);
            sprintf(DeltaX_text, "%.4lf", delta_X);
            sprintf(DeltaY_text, "%.4lf", delta_Y);

            X1->SetWindowText(X1_text);
            X2->SetWindowText(X2_text);
            Y1->SetWindowText(Y1_text);
            Y2->SetWindowText(Y2_text);
            Mag->SetWindowText(Mag_text);
            Dir->SetWindowText(Dir_text);
            DeltaX->SetWindowText(DeltaX_text);
            DeltaY->SetWindowText(DeltaY_text);
            ErrorNumber->SetWindowText(StrErrNum);
            ErrorType->SetWindowText(TheDlg-
>ErrorMessages[type]);

            TheDlg->InvalidateRect(&CRect(TheDlg->OriginX,
TheDlg->OriginY, TheDlg->OriginX+TheDlg->length, TheDlg-
>OriginY+TheDlg->length));
            TheDlg->InvalidateRect(&CRect(TheDlg-
>CompassOriginX, TheDlg->CompassOriginY, TheDlg-
>CompassOriginX+TheDlg->CompassLength, TheDlg-
>CompassOriginY+TheDlg->CompassLength));
            TheDlg->UpdateWindow();
            TheDlg->TrackDlg->RedrawWindow();
            //TheDlg->RedrawWindow();
        }
        Sleep(1000);
```

```
    }
  return 0;             // thread completed successfully
}

void CADTrackerDlg::OnButtonStop()
{
    // TODO: Add your control notification handler code here
    CONTINUE = FALSE;
}

void CADTrackerDlg::OnButtonQuit()
{
    // TODO: Add your control notification handler code here
    OnOK();

}

void CADTrackerDlg::OnButtonParameters()
{
    // TODO: Add your control notification handler code here
    ConstantsDlg->ShowWindow(SW_SHOW);
}

void CADTrackerDlg::OnButtonDialog()
{
    // TODO: Add your control notification handler code here
    CONTINUE = TRUE;
    AfxBeginThread(MultiThreadDebug, NULL);
}

UINT MultiThreadDebug( LPVOID pParam )
{
  /*CMyObject* pObject = (CMyObject*)pParam;

  if (pObject == NULL ||
     !pObject->IsKindOf(RUNTIME_CLASS(CMyObject)))
  return -1;            // illegal parameter
*/
    CADTrackerDlg* TheDlg = CADTrackerApp::GetDlg();

    delta_X=2.123456789;
    delta_Y=1.123456789;
    direction=(3.141592654/2);
    magnitude=4.0987654321;
    static int FIRSTTIME = 1;

    CStatic *X1 = (CStatic*)TheDlg->GetDlgItem(IDC_STATIC_X1);
    CStatic *X2 = (CStatic*)TheDlg->GetDlgItem(IDC_STATIC_X2);
    CStatic *Y1 = (CStatic*)TheDlg->GetDlgItem(IDC_STATIC_Y1);
    CStatic *Y2 = (CStatic*)TheDlg->GetDlgItem(IDC_STATIC_Y2);
    CStatic *Mag = (CStatic*)TheDlg->GetDlgItem(IDC_STATIC_MAG);
    CStatic *Dir = (CStatic*)TheDlg->GetDlgItem(IDC_STATIC_DIR);
    CStatic *DeltaX = (CStatic*)TheDlg-
>GetDlgItem(IDC_STATIC_DELTAX);
    CStatic *DeltaY = (CStatic*)TheDlg-
>GetDlgItem(IDC_STATIC_DELTAY);
    CStatic *ErrorNumber = (CStatic*)TheDlg-
>GetDlgItem(IDC_STATIC_NUMBER);
    CStatic *ErrorType = (CStatic*)TheDlg->GetDlgItem(IDC_STATIC_TYPE);


    char X1_text[32];
    char X2_text[32];
    char Y1_text[32];
    char Y2_text[32];
    char Mag_text[32];
    char Dir_text[32];
    char DeltaX_text[32];
    char DeltaY_text[32];
    char StrErrNum[8];

    int count=0;

    while(TheDlg->CONTINUE)
    {
        if(FIRSTTIME)
        {
            FIRSTTIME = 0;
            first.x = (double)2.0;
            first.y = (double)2.0;
            second.x = (double)4.0;
            second.y = (double)2.0;
        }

//      double MAX_RAND = 32565.0;

//      (double)rand()/MAX_RAND*4.0;
//      TheDlg->delta_velocity_X = (double)(rand()/MAX_RAND*4.0);
//      TheDlg->delta_velocity_Y = (double)(rand()/MAX_RAND*4.0);
//      TheDlg->direction = (double)(rand()/MAX_RAND*4.0);
//      TheDlg->magnitude = (double)(10.0*rand()/MAX_RAND*4.0);
//      TheDlg->first.x = (double)(rand()/MAX_RAND*4.0);
//      TheDlg->first.y = (double)(rand()/MAX_RAND*4.0);
//      TheDlg->second.x = (double)(rand()/MAX_RAND*4.0);
//      TheDlg->second.y = (double)(rand()/MAX_RAND*4.0);

        //Tracker Dialog Drawing Stuff
        TheDlg->magnitude_x = magnitude * cos(direction);
        TheDlg->magnitude_y = magnitude * sin(direction);

//      TRACE("mag,dir:  (%.4lf, %.4lf)\n", magnitude, direction);
        TRACE("magnitude_x,y = (%.4lf,%.4lf)\n", TheDlg->magnitude_x,
TheDlg->magnitude_y);

        int NewPt = TheDlg->NumPoints % 2048;
        int PrevPt = TheDlg->NumPoints % 2048;
        if(NewPt == 0)
            PrevPt = 2047;
        else
            PrevPt = NewPt - 1;

        TheDlg->ThePoints[NewPt].x = (int)((double)(TheDlg-
>ThePoints[PrevPt].x) + (TheDlg->magnitude_x) + 0.5);
        TheDlg->ThePoints[NewPt].y = (int)((double)(TheDlg-
>ThePoints[PrevPt].y) - (TheDlg->magnitude_y) + 0.5);

//      TRACE("(int)mag/dir==  (%d, %d)", ((int)(TheDlg->magnitude_x)),
((int)(TheDlg->magnitude_y)));
        TRACE("OldPoint(%d, %d) + (%.4lf,%.4lf) == NewPoint(%d,%d)\n",
TheDlg->ThePoints[PrevPt].x, TheDlg->ThePoints[PrevPt].y, TheDlg-
>magnitude_x, TheDlg->magnitude_y, TheDlg->ThePoints[NewPt].x, TheDlg-
>ThePoints[NewPt].y);
        TRACE("NP== (%d, %d)\n", ((int)(TheDlg->ThePoints[PrevPt].x +
(TheDlg->magnitude_x))), ((int)(TheDlg->ThePoints[PrevPt].y + (TheDlg-
>magnitude_y))));
        //TRACE("NP== (%d, %d)\n", ((int)(TheDlg->ThePoints[PrevPt].x +
(TheDlg->magnitude_x))), ((int)(TheDlg->ThePoints[PrevPt].y + (TheDlg-
>magnitude_y))));
        //TRACE("OldPt==%d, (%d,%d)  NewPt==%d, (%d,%d)\n", PrevPt,
TheDlg->ThePoints[PrevPt].x, TheDlg->ThePoints[PrevPt].y, NewPt, TheDlg-
>ThePoints[NewPt].x, TheDlg->ThePoints[NewPt].y );

        TheDlg->NumPoints++;
        if(TheDlg->NumPoints <= 2048)
            TheDlg->StartNumPoint = 0;
        else
            TheDlg->StartNumPoint = (NewPt+1)%2048;

        while(abs(TheDlg->ThePoints[(TheDlg->NumPoints-
1)%2048].x/TheDlg->TrackerScale) > TheDlg->TrackerCenterX + TheDlg-
>TrackerLength ||
              abs(TheDlg->ThePoints[(TheDlg->NumPoints-
1)%2048].y/TheDlg->TrackerScale) > TheDlg->TrackerCenterY + TheDlg-
>TrackerLength)
        {
            TheDlg->TrackerScale *= 2;
        }

        TRACE("TrackerScale==%d\n", TheDlg->TrackerScale);
        if(get_finger_debug())
        {
            TheDlg->MessageBox("I Failed on Run!\nExiting...");
            //OnOK();
        }
        else //Set all the Text Areas based on the new values
        {
            //TRACE("Run Number %d\n", count++);
            //TRACE("new:,  (%.3f, %.3f)  (%.3f,.%.3f) ", first.x,
first.y, old_first.x, old_first.y);

            sprintf(StrErrNum, "%d", type);
```

```
                    sprintf(X1_text, "%.3lf", first.x);
                    sprintf(X2_text, "%.3lf", second.x);
                    sprintf(Y1_text, "%.3lf", first.y);
                    sprintf(Y2_text, "%.3lf", second.y);
                    sprintf(Mag_text, "%.4lf", magnitude);
                    sprintf(Dir_text, "%.4lf", direction);
                    sprintf(DeltaX_text, "%.4lf", delta_X);
                    sprintf(DeltaY_text, "%.4lf", delta_Y);

                    X1->SetWindowText(X1_text);
                    X2->SetWindowText(X2_text);
                    Y1->SetWindowText(Y1_text);
                    Y2->SetWindowText(Y2_text);
                    Mag->SetWindowText(Mag_text);
                    Dir->SetWindowText(Dir_text);
                    DeltaX->SetWindowText(DeltaX_text);
                    DeltaY->SetWindowText(DeltaY_text);
                    ErrorNumber->SetWindowText(StrErrNum);
                    ErrorType->SetWindowText(TheDlg-
>ErrorMessages[type]);

                    TheDlg->InvalidateRect(&CRect(TheDlg->OriginX,
TheDlg->OriginY, TheDlg->OriginX+TheDlg->length, TheDlg-
>OriginY+TheDlg->length));
                    TheDlg->InvalidateRect(&CRect(TheDlg-
>CompassOriginX, TheDlg->CompassOriginY, TheDlg-
>CompassOriginX+TheDlg->CompassLength, TheDlg-
>CompassOriginY+TheDlg->CompassLength));
                    TheDlg->UpdateWindow();
                    TheDlg->TrackDlg->RedrawWindow();
                    //TheDlg->RedrawWindow();
            }
            Sleep(1000);
    }
    return 0;           // thread completed successfully
}
```

```
// ADTrackerDlg.h : header file

//


#ifndef __ADTRACKERDLG_H__

#define __ADTRACKERDLG_H__


/////////////////////////////////////////////////////////////////////////

// CADTrackerDlg dialog

#include "ModifyConstantsDlg.h"
#include "TrackerDlg.h"

class TrackerDlg;

class CADTrackerDlg : public CDialog
{
// Construction
public:
    CADTrackerDlg(CWnd* pParent = NULL);    // standard constructor
    ~CADTrackerDlg();
    // standard destructor

    BOOL CONTINUE;
/*  struct coord{
        double x;
        double y;
    } first, second;
    double delta_velocity_X;
    double delta_velocity_Y;
    double delta_X;
    double delta_Y;
    double direction;
    double magnitude;
*/  double magnitude_x;
    double magnitude_y;
    int OriginX;
    int OriginY;
    int CompassOriginX;
    int CompassOriginY;
    int CompassCenterX;
    int CompassCenterY;
    int CompassLength;
    int length;
    double MaxVal;
    //int type;
    char ErrorMessages[8][16];
    ModifyConstantsDlg* ConstantsDlg;
    TrackerDlg* TrackDlg;
    CPoint ThePoints[2048];
    int NumPoints;
    int StartNumPoint;
    int TrackerCenterX;
    int TrackerCenterY;
    int TrackerLength;
    int TrackerScale;

// Dialog Data
    //{{AFX_DATA(CADTrackerDlg)
    enum { IDD = IDD_ADTRACKER_DIALOG };
        // NOTE: the ClassWizard will add data members here
    //}}AFX_DATA

    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CADTrackerDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV
support
    //}}AFX_VIRTUAL

// Implementation
protected:
    HICON m_hIcon;

    // Generated message map functions
    //{{AFX_MSG(CADTrackerDlg)
    virtual BOOL OnInitDialog();
    afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
    afx_msg void OnPaint();
    afx_msg HCURSOR OnQueryDragIcon();
    afx_msg void OnButtonRun();
    afx_msg void OnButtonStop();
    afx_msg void OnButtonQuit();
    afx_msg void OnButtonParameters();
    afx_msg void OnButtonDialog();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

#endif // __ADTRACKERDLG_H__
```

```
// ModifyConstantsDlg.cpp : implementation file

//


#include "stdafx.h"

#include "ADTracker.h"

#include "ModifyConstantsDlg.h"

#include "finger.h"


#ifdef _DEBUG

#define new DEBUG_NEW

#undef THIS_FILE
```

```
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// ModifyConstantsDlg dialog


ModifyConstantsDlg::ModifyConstantsDlg(CWnd* pParent /*=NULL*/)
    : CDialog(ModifyConstantsDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(ModifyConstantsDlg)
        // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
    turning_multiplier = 1.0;
    forward_multiplier = 10.0;
    DebugMotion=0.0;
}


void ModifyConstantsDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(ModifyConstantsDlg)
        // NOTE: the ClassWizard will add DDX and DDV calls here
    //}}AFX_DATA_MAP
}


BEGIN_MESSAGE_MAP(ModifyConstantsDlg, CDialog)
    //{{AFX_MSG_MAP(ModifyConstantsDlg)
    ON_WM_SHOWWINDOW()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// ModifyConstantsDlg message handlers

void ModifyConstantsDlg::OnShowWindow(BOOL bShow, UINT nStatus)
{
    CDialog::OnShowWindow(bShow, nStatus);

    // TODO: Add your message handler code here
    CEdit* EditTurning = (CEdit*)GetDlgItem(IDC_EDIT_TURNING);
    CEdit* EditMagnitude = (CEdit*)GetDlgItem(IDC_EDIT_MAGNITUDE);
    CEdit* EditMotion = (CEdit*)GetDlgItem(IDC_EDIT_DEBUG_MOTION);

    char StrTurning[16];
    char StrMagnitude[16];
    char StrMotion[16];

    sprintf(StrTurning, "%.2lf", turning_multiplier);
    sprintf(StrMagnitude, "%.2lf", forward_multiplier);
    sprintf(StrMotion, "%.2lf", DebugMotion);

    EditTurning->SetWindowText(StrTurning);
    EditMagnitude->SetWindowText(StrMagnitude);
    EditMotion->SetWindowText(StrMotion);
}

void ModifyConstantsDlg::OnOK()
{
    // TODO: Add extra validation here
    CEdit* EditTurning = (CEdit*)GetDlgItem(IDC_EDIT_TURNING);
    CEdit* EditMagnitude = (CEdit*)GetDlgItem(IDC_EDIT_MAGNITUDE);
    CEdit* EditMotion = (CEdit*)GetDlgItem(IDC_EDIT_DEBUG_MOTION);

    char StrTurning[32];
    char StrMagnitude[32];
    char StrMotion[32];

    EditTurning->GetWindowText(StrTurning, 31);
    EditMagnitude->GetWindowText(StrMagnitude, 31);
    EditMotion->GetWindowText(StrMotion, 31);

    turning_multiplier = strtod(StrTurning, NULL);
    forward_multiplier = strtod(StrMagnitude, NULL);
    DebugMotion = strtod(StrMotion, NULL);

    CDialog::OnOK();
}
```

```
void ModifyConstantsDlg::OnCancel()
{
    // TODO: Add extra cleanup here

    CDialog::OnCancel();
}
```

```
#if
!defined(AFX_MODIFYCONSTANTSDLG_H__44525C02_91A3_11D1_8C94_
004005368232__INCLUDED_)

#define
AFX_MODIFYCONSTANTSDLG_H__44525C02_91A3_11D1_8C94_00400536
8232__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
// ModifyConstantsDlg.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// ModifyConstantsDlg dialog

class ModifyConstantsDlg : public CDialog
{
// Construction
public:
    ModifyConstantsDlg(CWnd* pParent = NULL);  // standard constructor

// Dialog Data
    //{{AFX_DATA(ModifyConstantsDlg)
    enum { IDD = IDD_DIALOG_CONSTANTS };
        // NOTE: the ClassWizard will add data members here
    //}}AFX_DATA

/*  double turning_multiplier;
    double forward_multiplier;
    double DebugMotion;
*/
// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(ModifyConstantsDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV
support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(ModifyConstantsDlg)
    virtual void OnOK();
    virtual void OnCancel();
    afx_msg void OnShowWindow(BOOL bShow, UINT nStatus);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately
before the previous line.

#endif //
!defined(AFX_MODIFYCONSTANTSDLG_H__44525C02_91A3_11D1_8C94_
004005368232__INCLUDED_)
```

```
//{{NO_DEPENDENCIES}}

// Microsoft Developer Studio generated include file.

// Used by ADTracker.rc
```

```
//

#define IDM_ABOUTBOX                    0x0010

#define IDD_ABOUTBOX                    100
#define IDS_ABOUTBOX                    101
#define IDD_ADTRACKER_DIALOG            102
#define IDR_MAINFRAME                   128
#define IDR_MENU1                       129
#define IDD_DIALOG_CONSTANTS            130
#define IDD_DIALOG_TRACKER              131
#define IDC_BUTTON_RUN                  1000
#define IDC_BUTTON_STEP                 1001
#define IDC_BUTTON_DIALOG               1001
#define IDC_BUTTON_STOP                 1002
#define IDC_BUTTON_PARAMETERS           1003
#define IDC_BUTTON_QUIT                 1004
#define IDC_STATIC_COORDINATES          1005
#define IDC_STATIC_VELOCITY             1006
#define IDC_STATIC_X2                   1007
#define IDC_STATIC_X1                   1008
#define IDC_STATIC_Y1                   1009
#define IDC_STATIC_Y2                   1010
#define IDC_STATIC_MAG                  1013
#define IDC_STATIC_DIR                  1014
#define IDC_STATIC_DELTAY               1015
#define IDC_STATIC_DELTAX               1016
#define IDC_STATIC_TYPE                 1017
#define IDC_STATIC_NUMBER               1018
#define IDC_STATIC_SPEC_INST            1019
#define IDC_EDIT_TURNING                1020
#define IDC_STATIC_TURN_MULT            1021
#define IDC_STATIC_MAG_MULT             1022
#define IDC_EDIT_MAGNITUDE              1023
#define IDC_STATIC_Debug1               1024
#define IDC_EDIT_DEBUG_MOTION           1025
#define ID_FILE_RUN                     32771
#define ID_FILE_STOP                    32772

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        132
#define _APS_NEXT_COMMAND_VALUE         32773
#define _APS_NEXT_CONTROL_VALUE         1025
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif
```

```
// stdafx.cpp : source file that includes just the standard includes

//    ADTracker.pch will be the pre-compiled header

//    stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"
```

```
// stdafx.h : include file for standard system include files,

//  or project specific include files that are used frequently, but

//      are changed infrequently

//



#define VC_EXTRALEAN                    // Exclude rarely-used stuff from
Windows headers

#include <afxwin.h>         // MFC core and standard components
#include <afxext.h>         // MFC extensions
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>                     // MFC support for Windows 95
Common Controls
#endif // _AFX_NO_AFXCMN_SUPPORT
```

```
// TrackerDlg.cpp : implementation file

//



#include "stdafx.h"

#include "ADTracker.h"

#include "TrackerDlg.h"



#ifdef _DEBUG

#define new DEBUG_NEW

#undef THIS_FILE

static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// TrackerDlg dialog

TrackerDlg::TrackerDlg(CWnd* pParent /*=NULL*/)
    : CDialog(TrackerDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(TrackerDlg)
        // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
    Parent = (CADTrackerDlg*)pParent;
}


void TrackerDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(TrackerDlg)
        // NOTE: the ClassWizard will add DDX and DDV calls here
    //}}AFX_DATA_MAP
}


BEGIN_MESSAGE_MAP(TrackerDlg, CDialog)
    //{{AFX_MSG_MAP(TrackerDlg)
    ON_WM_PAINT()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// TrackerDlg message handlers

void TrackerDlg::OnPaint()
{
    CPaintDC dc(this); // device context for painting

    // TODO: Add your message handler code here
    int i=0;
    dc.MoveTo(Parent->TrackerCenterX, Parent->TrackerCenterY);
    if(Parent->NumPoints > 1)
        for(i=Parent->StartNumPoint; i < Parent->StartNumPoint + min(2048,
Parent->NumPoints); i++)
        {
                TRACE("Drawing from (%d,%d) to (%d,%d)\n",
dc.GetCurrentPosition().x, dc.GetCurrentPosition().y, Parent-
>ThePoints[i%2048].x/Parent->TrackerScale, Parent-
>ThePoints[i%2048].y/Parent->TrackerScale);
                dc.LineTo(Parent->ThePoints[i%2048].x/Parent-
>TrackerScale, /*Parent->TrackerLength - */Parent-
>ThePoints[i%2048].y/Parent->TrackerScale);
        }
```

```
    // Do not call CDialog::OnPaint() for painting messages
}
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

#if
!defined(AFX_TRACKERDLG_H__44525C06_91A3_11D1_8C94_004005368232__INCLUDED_)

#define
AFX_TRACKERDLG_H__44525C06_91A3_11D1_8C94_004005368232__INCLUDED_


#if _MSC_VER >= 1000

#pragma once

#endif // _MSC_VER >= 1000
// TrackerDlg.h : header file
//

#include "ADTrackerDlg.h"
class CADTrackerDlg;

/////////////////////////////////////////////////////////////////////////////
// TrackerDlg dialog

class TrackerDlg : public CDialog
{
// Construction
public:
    TrackerDlg(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
    //{{AFX_DATA(TrackerDlg)
    enum { IDD = IDD_DIALOG_TRACKER };
        // NOTE: the ClassWizard will add data members here
    //}}AFX_DATA

    CADTrackerDlg *Parent;

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(TrackerDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV
support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(TrackerDlg)
    afx_msg void OnPaint();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately
before the previous line.

#endif //
!defined(AFX_TRACKERDLG_H__44525C06_91A3_11D1_8C94_004005368232__INCLUDED_)
```

References

Bechwith, Marangoni, and Lienhard (1993). Mechanical Measurements. Addison-Wesley Publishing Company. Massachusetts.

Bik, Russell (1997). Electric Field Proximity Sensor. Nuts & Volts Magazine.

Durlach and Mavor (1995). Virtual Reality: Scientific And Technical Challenges. National Academy Press.

Isdale Engineering (1997). What Is Virtual Reality?. World Wide Web http://www.columbia.edu/%7Erk35/vr/vr.html

Iwata, Hiroo (1992). Force Displays for Walkthrough Simulation. Proceedings of ISMCR'92. Tsukuba Science City, Japan. 481-486.

Iwata, Hiroo and Matsuda, Keigo (1992). Haptic Walkthrough Simulator: Its Design And Application To Studies On Cognitive Map. Proceedings of ICAT '92. 185-192.

Iwata, Hiroo and Fujii, Takashi (1996). Virtual Perambulator: A Novel Interface Device for Locomotion In Virtual Environments. Proceedings of VRAIS'96. 60-65.

Jacobsen, S., & Collard, B. (1996). Characterization of Mobility Platforms for Interfacing Humans to Virtual Environments. Sarcos Research Corporation.

Koh, Glenn (1997). Training Spatial Knowledge Acquisition Using Virtual Environments. Master's Thesis. Department of Electrical Engineering and Computer Science. MIT.

Mississippi State University (1997). An Investigation Of Current Virtual Reality Interfaces. World Wide Web http://www2.msstate.edu/~cms2/hcifinal.html

Slater, Usoh, and Steed (1995). Taking Steps: The Influence of a Walking Technique on Presence in Virtual Reality. ACM Transactions on Computer-Human Interaction, 2(3).

Slater, Mel and Wilbur, Sylvia (1997). A Framework for Immersive Virtual Environments (FIVE): Speculations on the Role of Presence in Virtual Environments. Presence, 6(6).

Smith, J. R. (1995). Towards Electric Field Tomography. Master's Thesis. Department of Media Arts and Sciences. MIT.

Smith, J. R. (1996). Field mice: Extracting Hand Geometry from Electric Field Measurements. IBM Systems Journal, 35(3&4).

Tan, Durlach, Beauregard, Srinivasan (1995). Manual Discrimination of Compliance Using Active Pinch Grasp: The Roles of Force and Work Cues. Psychonomic Society, Inc. 57(4).

Templeman, James (1998). Performance Based Design of a New Virtual Locomotion Control. Unpublished.

Virtual Environment Technology for Training (1996). Training Spatial Knowledge Acquisition Using Virtual Environments. World Wide Web http://mimsy.mit.edu/people/sploc/sploc.html

Wies, Evan (1996). The Addition of the Haptic Modality to the Virtual Reality Modeling Language. Master's Thesis. Department of Electrical Engineering and Computer Science. MIT.